Bisimulation and coinduction

Davide Sangiorgi

Focus Team,
INRIA (France)/University of Bologna (Italy)

Email: Davide.Sangiorgi@cs.unibo.it
http://www.cs.unibo.it/~sangio/

BiSS 2010, Bertinoro, March 2010

Objectives of the course

At the end of the course, a student should:

- have understood what bisimulation and coinduction are
- be able to do (simple) bisimulation proofs and write coinductive definitions
- have a grasp of the duality between induction and coinduction
- know what a behavioural equivalence is, in particular the difference between bisimilarity and other behavioural equalities

References

This course is based on the draft book:

- Davide Sangiorgi, *An introduction to bisimulation and coinduction*, Draft, 2010

Please contact me if you'd like to read and comment parts of it.

Exam

- To be discussed
- Based on chapters of the book

Outline

- The success of bisimulation and coinduction
- Towards bisimulation, or: from functions to processes
- Bisimulation
- Induction and coinduction
- Weak bisimulation
- Other equivalences: failures, testing, trace ...
- Enhancements of the bisimulation proof method
- ....

The success of bisimulation and coinduction

Bisimulation in Computer Science

- One of the most important contributions of concurrency theory to CS
- It has spurred the study of coinduction
- In concurrency: probably the most studied equivalence
  - ... in a plethora of equivalences **(see van Glabbeek 93)**
  - Why?

Bisimulation in concurrency

- **Clear** meaning of equality
- **Natural**
- The **finest** extensional equality

  Extensional: – "whenever it does an output at b it will also do an input at a"

  Non-extensional: – "Has 8 states"
  – "Has an Hamiltonian circuit"
- An associated powerful **proof technique**
- **Robust**

  Characterisations: logical, algebraic, set-theoretical, categorical, game-theoretical, ...
- Several **separation results** from other equivalences

Bisimulation in concurrency, today

- To **define equality** on processes (fundamental !!)
- To **prove equalities**
  - even if bisimilarity is not the chosen equivalence
    - trying bisimilarity first
    - coinductive characterisations of the chosen equivalence
- To **justify algebraic laws**
- To **minimise** the state space
- To **abstract** from certain details

Coinduction in programming languages

- **Bisimilarity in functional languages and OO languages**

  [Abramsky, Ong]

  A major factor in the movement towards operationally-based techniques in PL semantics in the 90s

- **Program analysis** **(see Nielson, Nielson, Hankin 's book)**

  Noninterference (security) properties

- **Verification tools**: algorithms for computing gfp (for modal and temporal logics), tactics and heuristics

- **Types** [Tofte]
  - type soundness
  - coinductive types and definition by corecursion
    Infinite proofs in Coq [Coquand, Gimenez]
  - recursive types (equality, subtyping, ...)
    A coinductive rule:

$$\frac{\Gamma, \langle p_1, q_1 \rangle \sim \langle p_2, q_2 \rangle \vdash p_i \sim q_i}{\Gamma \vdash \langle p_1, q_1 \rangle \sim \langle p_2, q_2 \rangle}$$

- **Recursively defined data types and domains** [Fiore, Pitts]
- **Databases** [Buneman]
- **Compiler correctness** [Jones]

Towards bisimulation, or:

from functions to processes

The semantics of processes:

- usually **operational**: (Labelled Transitions Systems, behavioural equivalences)
- alternative approach could be the **denotational** one: a structure-preserving function would map processes into elements of a given semantic domain.
  Problem: it has often proved very hard to find appropriate semantic domains for these languages

Processes?

We can think of sequential computations as mathematical objects, namely **functions**.

Concurrent program are not functions, but **processes**. But what is a process?

No universally-accepted mathematical answer.

Hence we do not find in mathematics tools/concepts for the denotational semantics of concurrent languages, at least not as successful as those for the sequential ones.

Processes are not functions

A sequential imperative language can be viewed as a function from states to states.

These two programs denote the same function from states to states:

$$\mathtt{X} := 2 \qquad \text{and} \qquad \mathtt{X} := 1;\ \mathtt{X} :=\ \mathtt{X} + 1$$

But now take a context with parallelism, such as $[\cdot] \mid\ \mathtt{X} := 2$. The program

$$\mathtt{X} := 2 \mid\ \mathtt{X} := 2$$

always terminates with $\mathtt{X} = 2$. This is not true (why?) for

$$(\ \mathtt{X} := 1;\ \mathtt{X} :=\ \mathtt{X} + 1\ ) \mid\ \mathtt{X} := 2$$

Therefore: Viewing processes as functions gives us a notion of equivalence that is not a **congruence**. In other words, such a semantics of processes as functions would not be **compositional**.

Furthermore:

- A concurrent program may not terminate, and yet perform meaningful computations (examples: an operating system, the controllers of a nuclear station or of a railway system).
  In sequential languages programs that do not terminate are undesirable; they are ‘wrong’.

- The behaviour of a concurrent program can be non-deterministic.
  Example:
  $$(\ \mathtt{X} := 1;\ \ \mathtt{X} :=\ \ \mathtt{X} + 1\ ) \mid\ \ \mathtt{X} := 2$$
  In a functional approach, non-determinism can be dealt with using powersets and powerdomains.

This works for pure non-determinism, as in $\lambda x.\,(3 \oplus 5)$

But not for parallelism.

What is a process?
When are two processes behaviourally equivalent?

These are basic, fundamental, questions; they have been at the core of the research in concurrency theory for the past 30 years. (They are still so today, although remarkable progress has been made)

Fundamental for a model or a language on top of which we want to make proofs ...

We shall approach these questions from a simple case, in which interactions among processes are just synchronisations, without exchange of values.

Interaction

In the example at page 14

$$\mathtt{x} := 2 \qquad \text{and} \qquad \mathtt{x} := 1;\ \mathtt{x} := \mathtt{x} + 1$$

should be distinguished because they interact in a different way with the memory.

Computation is **interaction**. Examples: access to a memory cell, interrogating a data base, selecting a programme in a washing machine, ...

The participants of an interaction are **processes** (a cell, a data base, a washing machine, ...)

The **behaviour** of a process should tell us **when** and **how** a process can interact with its environment

How to represent interaction: labelled transition systems

Definition 1 A **labeled transition system** (LTS) is a triple $(\mathcal{P}, \mathtt{Act}, \mathcal{T})$ where

- $\mathcal{P}$ is the set of **states**, or **processes**;
- $\mathtt{Act}$ is the set of **actions**; (NB: can be infinite)
- $\mathcal{T} \subseteq (\mathcal{P}, \mathtt{Act}, \mathcal{P})$ is the **transition relation**.

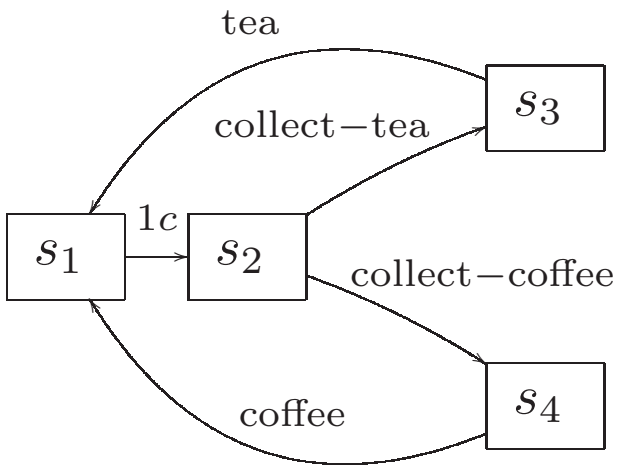
We write $P \xrightarrow{\mu} P'$ if $(P, \mu, P') \in \mathcal{T}$. Meaning: process P accepts an interaction with the environment where P performs action μ and then becomes process P'.

P' is a **derivative** of P if there are $P_1, \ldots, P_n, \mu_1, \ldots, \mu_n$ s.t. $P \xrightarrow{\mu_1} P_1 \ldots \xrightarrow{\mu_n} P_n$ and $P_n = P'$.

Example

A vending machine, capable of dispensing tea or coffee for 1 coin (1c).

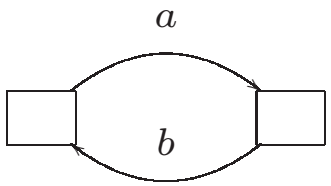
The behaviour of the machine is what we can observe, by interacting with the machine. We can represent such a behaviour as an LTS:

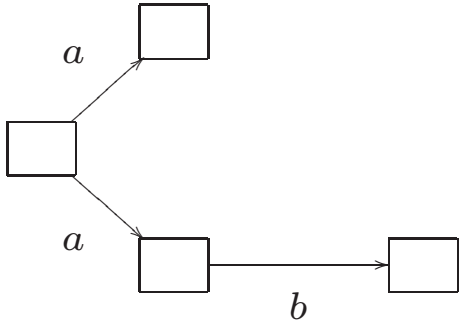


(where s_1 is the initial state)

Other examples of LTS

(we omit the name of the states)





Equivalence of processes

An LTS tells us what is the behaviour of processes. When should two behaviours be considered equal? ie, what does it mean that two processes are equivalent?

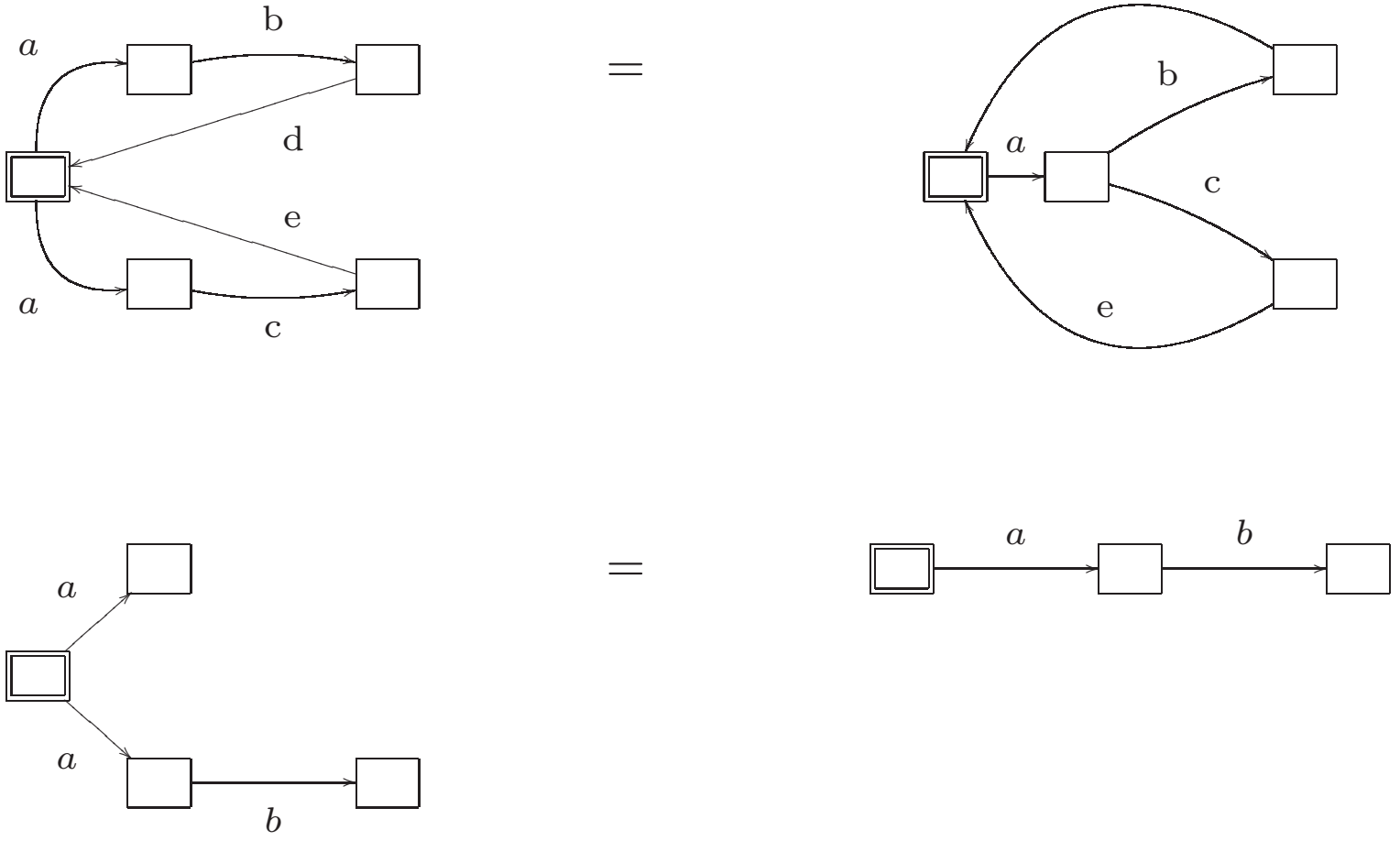
Two processes should be equivalent if we cannot distinguish them by interacting with them.

Example (where ▭ indicates the processes we are interested in):

This shows that **graph isomorphism** as behavioural equivalence is too strong.

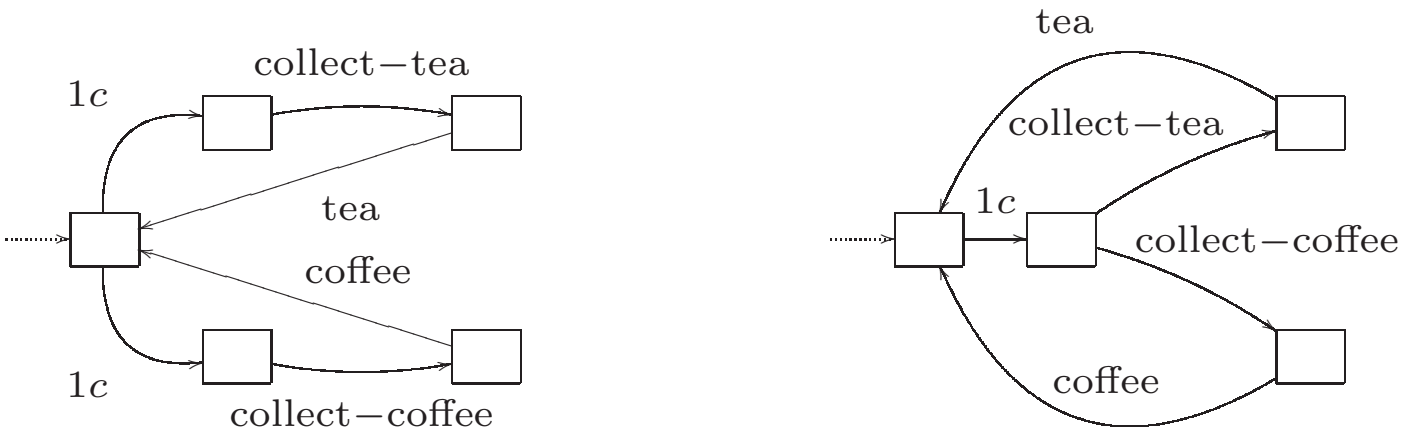
A natural alternative (from automata theory): **trace equivalence**.

Examples of trace-equivalent processes:



These equalities are OK on automata. But they are not on processes: in one case interacting with the machine can lead to deadlock!

For instance, you would not consider these two vending machines ‘the same’:



Trace equivalence (also called language equivalence) is still important in concurrency.

Examples: confluent processes; liveness properties such as termination

These examples suggest that the notion of equivalence we seek:

- should imply a tighter correspondence between transitions than language equivalence,
- should be based on the informations that the transitions convey, and not on the shape of the diagrams.

Intuitively, what does it mean for an observer that two machines are equivalent?

If you do something with one machine, you must be able to the same with the other, and on the two states which the machines evolve to the same is again true.

This is the idea of equivalence that we are going to formalise; it is called **bisimilarity**.

Bisimulation and bisimilarity

We define bisimulation on a single LTS, because: the union of two LTSs is an LTS; we will often want to compare derivatives of the same process.

Definition 2 (bisimulation) A relation $\mathcal{R}$ on the states of an LTS is a **bisimulation** if whenever $P \mathcal{R} Q$:

1. $\forall \mu, P'$ s.t. $P \xrightarrow{\mu} P'$, then $\exists Q'$ such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$;
2. $\forall \mu, Q'$ s.t. $Q \xrightarrow{\mu} Q'$, then $\exists P'$ such that $P \xrightarrow{\mu} P'$ and $P' \mathcal{R} Q'$.

P and Q are **bisimilar**, written $P \sim Q$, if $P \mathcal{R} Q$, for some bisimulation $\mathcal{R}$.

The bisimulation diagram:

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ \mu \downarrow & & \mu \downarrow \\ P' & \mathcal{R} & Q' \end{array}$$

Exercises

To prove $P \sim Q$ you have to find a bisimulation $\mathcal{R}$ with $P \; \mathcal{R} \; Q$ (the *bisimulation proof method*)

Exercise 1 Prove that the processes at page 21 are bisimilar. Are the processes at page 22 bisimilar?

Proposition 1 1. $\sim$ is an equivalence relation, i.e. the following hold:

1.1. $P \sim P$ (reflexivity)

1.2. $P \sim Q$ implies $Q \sim P$ (symmetry)

1.3. $P \sim Q$ and $Q \sim R$ imply $P \sim R$ (transitivity);

2. $\sim$ itself is a bisimulation.

Proposition 1(2) suggests an alternative definition of $\sim$:

Proposition 2 $\sim$ is the largest relation among the states of the LTS such that $P \sim Q$ implies:

1. $\forall \mu, P'$ s.t. $P \xrightarrow{\mu} P'$, then $\exists Q'$ such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$;
2. $\forall \mu, Q'$ s.t. $Q \xrightarrow{\mu} Q'$, then $\exists P'$ such that $P \xrightarrow{\mu} P'$ and $P' \sim Q'$.

Exercise 2 Prove Propositions 1-2
(for 1(2) you have to show that

$$\cup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation } \}$$

is a bisimulation).

We write $P \sim\mathcal{R}\sim Q$ if there are P', Q' s.t. $P \sim P'$, $P' \mathcal{R} Q'$, and $Q' \sim Q$ (and alike for similar notations).

Definition 3 (bisimulation up-to $\sim$) A relation $\mathcal{R}$ on the states of an LTS is a *bisimulation up-to* $\sim$ if $P \mathcal{R} Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim\mathcal{R}\sim Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim\mathcal{R}\sim Q'$.

Exercise 3 If $\mathcal{R}$ is a bisimulation up-to $\sim$ then $\mathcal{R} \subseteq \sim$. (Hint: prove that $\sim \mathcal{R} \sim$ is a bisimulation.)

Definition 4 (simulation) A relation $\mathcal{R}$ on the states of an LTS is a *simulation* if $P \mathcal{R} Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$.

P is *simulated by* Q, written $P < Q$, if $P \mathcal{R} Q$, for some simulation $\mathcal{R}$.

Exercise* 1 Does $P \sim Q$ imply $P < Q$ and $Q < P$? What about the converse? (Hint for the second point: think about the 2nd equality at page 22.)

Exercise 4 Let μ^+ range over non-empty sequences of actions. Consider the following definition of bisimulation:

A relation $\mathcal{R}$ on the states of an LTS is a *bisimulation* if $P \ \mathcal{R} \ Q$ implies:

1. if $P \xrightarrow{\mu^+} P'$, then there is Q' such that $Q \xrightarrow{\mu^+} Q'$ and $P' \ \mathcal{R} \ Q'$;
2. if $Q \xrightarrow{\mu^+} Q'$, then there is P' such that $P \xrightarrow{\mu^+} P'$ and $P' \ \mathcal{R} \ Q'$.

Prove that the definition above is the same as Definition 2.

- Bisimulation has been introduced in Computer Science by Park (1981) and made popular by Milner.

- Bisimulation is a robust notion: characterisations of bisimulation have been given in terms of non-well-founded-sets, modal logic, final coalgebras, open maps in category theory, etc.

- But the most important feature of bisimulation is the associated **coinductive** proof technique.

Induction and coinduction

coinductive definitions and coinductive proofs

Bisimulation:

A relation $\mathcal{R}$ s.t.

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\alpha} \\ P' & \mathcal{R} & Q' \end{array}$$

Bisimilarity ($\sim$) :

$$\bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation } \}$$

Hence:

$$\frac{x\ \mathcal{R}\ y \qquad \mathcal{R} \text{ is a bisimulation}}{x \sim y}$$

(bisimulation proof method)

– The definition of $\sim$ seems circular

(From Proposition 2)

$\sim$ is the largest relation such that $P \sim Q$ implies:

(1) $\forall \mu, P'$ s.t. $P \xrightarrow{\mu} P'$, then
$\exists Q'$ such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$;

(2) $\forall \mu, Q'$ s.t. $Q \xrightarrow{\mu} Q'$, then
$\exists P'$ such that $P \xrightarrow{\mu} P'$ and $P' \sim Q'$.

does it make sense?

– We claimed that we can prove $(P, Q) \in \sim$ by showing that $(P, Q) \in \mathcal{R}$ and $\mathcal{R}$ is a *bisimulation relation*, that is a relation that satisfies the same clauses as $\sim$. Does such a proof technique make sense?

– Contrast all this with the usual, familiar *inductive definitions* and *inductive proofs*.

– The definition of $\sim$, and the associated proof technique are examples of a *coinductive definition* and of a *coinductive proof technique*.

Bisimulation and coinduction: what are we talking about?
Has *co*-induction anything to do with induction?

Examples of induction and coinduction

An inductive definition: finite lists over a set A

$$\frac{}{\texttt{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \qquad a \in A}{\langle a \rangle \bullet \ell \in \mathcal{L}}$$

Inductively, the rules are read in the **"forward"** direction

3 equivalent readings:

- The objects obtained with a finite proof from the rules
- (iterative construction) Start from $\emptyset$; add all objects as in the axiom; repeat adding objects following the inference rule forwards
- The smallest set closed forward under these rules

A set T is closed forward if:
- $\texttt{nil} \in T$
- $\ell \in T$ implies $\langle a \rangle \bullet \ell \in T$, for all $a \in A$

Inductive proof technique for lists: Let T be a predicate (a property) on lists. To prove that T holds on all lists, prove that T is closed forward

A coinductive definition: finite and infinite lists over A

$$\frac{}{\texttt{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \qquad a \in A}{\langle a \rangle \bullet \ell \in \mathcal{L}}$$

Coinductively, the rules are read in the **"backward"** direction

3 equivalent readings:

- The objects that are conclusion of a finite or infinite proof from the rules
- $X =$ all (finite and infinite) strings of $A \cup \{\texttt{nil}, \langle, \rangle, \bullet\}$
  Start from X (all strings) and keep removing strings, following the backward-closure
- The largest set closed backward under these rules

A set T is closed backward if $\forall t \in T$:
– either $t = \texttt{nil}$
– or $t = \langle a \rangle \bullet \ell$, for some $\ell \in T$ and $a \in A$

Coinduction proof method: to prove that ℓ is a finite or infinite list, find a set D with $\ell \in D$ and D closed backwards

An example of an inductive definition: finite traces

A process P is **stopped** if it cannot do any transitions (i.e., $P \stackrel{\mu}{\not\longrightarrow}$ for all μ).

P has a **a finite trace**, written $P \downarrow$, if P has a finite sequence of transitions that lead to a stopped process

$\downarrow$ has a natural inductive definition:

$$\frac{P \text{ stopped}}{P \downarrow} \qquad \frac{P \stackrel{\mu}{\longrightarrow} P' \qquad P' \downarrow}{P \downarrow}$$

$\downarrow$ is the **smallest** set of processes that is **closed forward under the rules**; i.e., the smallest subset S of *Pr* (all processes) such that

- all stopped processes are in S;
- if there is μ such that $P \stackrel{\mu}{\longrightarrow} P'$ for some $P' \in S$, then also $P \in S$.

Constructively: you can start from $\emptyset$ and keep adding processes, following the forward-closure, until no more processes can be added

An example of a coinductive definition: ω-traces

P has an **ω-trace under μ**, written $P \upharpoonright_\mu$, if it is possible to observe an infinite sequence of μ-transitions starting from P.

$\upharpoonright_\mu$ has a natural **coinductive** definition in terms of rules:

$$\frac{P \xrightarrow{\mu} P' \qquad P' \upharpoonright_\mu}{P \upharpoonright_\mu}$$

$\upharpoonright_\mu$ is the **largest** predicate on processes that is **closed backward under the rule**; i.e., the largest subset S of processes such that if $P \in S$ then

- there is $P' \in S$ such that $P \xrightarrow{\mu} P'$.

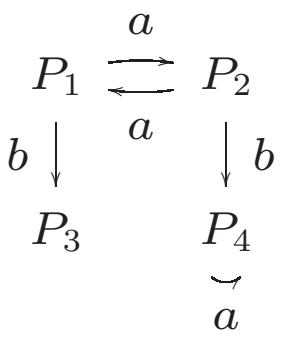
Constructively: you can start from *Pr* (all processes) and keep removing processes, following the backward-closure, until no more processes can be removed

Hence: to prove that a given process P has an ω-trace under μ it suffices to find some $T \subseteq \mathit{Pr}$ that is closed backward and with $P \in T$;

This is the coinduction proof principle, for ω-traces.

What is the smallest predicate closed backwards?

Example:



The set $S_1 \stackrel{\text{def}}{=} \{P_1, P_2\}$ is closed backward under the rules for $\restriction_a$, hence $P_1 \restriction_a$ and $P_2 \restriction_a$ hold.

Other such sets are $S_2 = \{P_4\}$ and $S_1 \cup S_2$.

Note that on the processes P_1 and P_2 both $\downarrow$ and $\restriction_a$ hold.

An inductive definition: convergence, in λ-calculus

Set of λ-terms (an inductive def!) $\quad e ::= x \mid \lambda x.\, e \mid e_1(e_2)$

Convergence to a value ($\Downarrow$), on closed λ-terms, call-by-name:

$$\frac{}{\lambda x.\, e \Downarrow \lambda x.\, e} \qquad \frac{e_1 \Downarrow \lambda x.\, e_0 \qquad e_0\{e_2/x\} \Downarrow e'}{e_1(e_2) \Downarrow e'}$$

As before, $\Downarrow$ can be read in terms of finite proofs, limit of an iterative construction, or smallest set closed forward under these rules

$\Downarrow$ is the smallest relation $\mathcal{S}$ on (closed) λ-terms s.t.
- $\lambda x.\, e \; C \; \lambda x.\, e$ for all abstractions,
- if $e_1 \; C \; \lambda x.\, e_0$ and $e_0\{e_2/x\} \; C \; e'$ then also $e_1(e_2) \; C \; e'$.

A coinductive definition: divergence in the λ-calculus

Divergence ($\Uparrow$), on closed λ-terms, call-by-name:

$$\frac{e_1 \Uparrow}{e_1(e_2) \Uparrow} \qquad \frac{e_1 \Downarrow \lambda x.\, e_0 \qquad e_0\{e_2/x\} \Uparrow}{e_1(e_2) \Uparrow}$$

The 'closed backward' reading:

> $\Uparrow$ is the *largest* predicate on λ-terms that is closed backward under these rules; i.e., the largest subset D of λ-terms s.t. if $e \in D$ then
> – either $e = e_1(e_2)$ and $e_1 \in D$,
> – or $e = e_1(e_2)$, $e_1 \Downarrow \lambda x.\, e_0$ and $e_0\{e_2/x\} \in D$.

Coinduction proof technique :

to prove $e \Uparrow$, find $E \subseteq \Lambda$ closed backward and with $e \in E$

What is the smallest predicate closed backwards?

The duality induction/coinduction

Constructors/destructors

- An inductive definition tells us what are the **constructors** for generating all the elements (cf: the forward closure).
- A coinductive definition tells us what are the **destructors** for decomposing the elements (cf: the backward closure).

The destructors show what we can **observe** of the elements
(think of the elements as black boxes;
the destructors tell us what we can do with them;
this is clear in the case of infinite lists).

Definitions given by means of rules

- if the definition is **inductive**, we look for the **smallest** universe in which such rules live.
- if it is **coinductive**, we look for the **largest** universe.
- the **inductive proof principle** allows us to infer that the **inductive set is included in a set** (ie, has a given property) by proving that the set satisfies the **forward closure**;
- the **coinductive proof principle** allows us to infer that **a set is included in the coinductive set** by proving that the given set satisfies the **backward closure**.

Forward and backward closures

A set T being closed forward intuitively means that

> *for each* **rule whose premise is satisfied in** T
> *there is* **an element of** T
> *such that the element is the conclusion of the rule.*

In the backward closure for T, the order between the two quantified entities is swapped:

> *for each* **element of** T
> *there is* **a rule whose premise is satisfied in** T
> *such that the element is the conclusion of the rule.*

In fixed-point theory, the duality between forward and backward closure will the duality between pre-fixed points and post-fixed points.

Congruences vs bisimulation equivalences

Congruence: an equivalence relation
that respects the constructors of a language

Example (λ-calculus)

Consider the following rules, acting on pairs of (open) λ-terms:

$$\frac{}{(x, x)} \qquad \frac{(e_1, e_2)}{(e\ e_1, e\ e_2)} \qquad \frac{(e_1, e_2)}{(e_1\ e, e_2\ e)} \qquad \frac{(e_1, e_2)}{(\lambda x.\, e_1, \lambda x.\, e_2)}$$

A congruence: an equivalence relation closed forward under the rules

The smallest such relation is **syntactic equality**: the **identity relation**

In other words, congruence rules express syntactic constraints

Bisimulation equivalence: an equivalence relation that respects the destructors

Example (λ-calculus, call-by-name)

Consider the following rules

$$\frac{e_1 \Uparrow \qquad e_2 \Uparrow}{(e_1, e_2)} \quad e_1, e_2 \text{ closed}$$

$$\frac{e_1 \Downarrow \lambda x.\, e_1' \qquad e_2 \Downarrow \lambda x.\, e_2' \qquad \cup_{e''} \{(e_1'\{e''/x\}, e_2'\{e''/x\})\}}{(e_1, e_2)} \quad e_1, e_2, e'' \text{ closed}$$

$$\frac{\cup_{\sigma} \{(e_1\sigma, e_2\sigma)\}}{(e_1, e_2)} \quad e_1, e_2 \text{ non closed, } \sigma \text{ closing substitution for } e_1, e_2$$

A bisimulation equivalence: an equivalence relation closed backward under the rules

The largest such relation is **semantic equality**: **bisimilarity**

In other words, the bisimulation rules express semantic constraints

Substitutive relations vs bisimulations

In the duality between congruences and bisimulation equivalences, the equivalence requirement is not necessary.

Leave it aside, we obtaining the duality between **bisimulations** and **substitutive relations**

> a relation is substitutive if whenever s and t are related,
> then any term t' must be related to a term s'
> obtained from t' by replacing occurrences of t with s

Bisimilarity is a congruence

To be useful, a bisimilarity on a term language should be a congruence

This leads to proofs where inductive and coinductive techniques are intertwined

In certain languages, for instance higher-order languages, such proofs may be hard, and how to best combine induction and coinduction remains a research topic.

What makes the combination delicate is that the rules on which congruence and bisimulation are defined — the rules for syntactic and semantic equality — are different.

The duality

inductive definition	coinductive definition
induction proof principle	coinduction proof principle
constructors	observations
smallest universe	largest universe
least fixed point	greatest fixed point
pre-fixed points	post-fixed point
’forward closure’ in rules	’backward closure’ in rules
congruence	bisimulation equivalence
substitutive relation	bisimulation
identity	bisimilarity
syntax	semantics

- In what sense are $\sim$, $\lceil_\mu$, $\lfloor$, etc. fixed-points?
- What is exactly the coinduction proof technique?
- What is the mathematical sense of the duality between induction and coinduction ?

What follows answers these questions. It is a simple application of fixed-point theory on complete lattices.

To make things simpler, we work on *powersets* and *fixed-point theory*. (It is possible to be more general, working with universal algebras or category theory.)

Complete lattices

The important example of complete lattice for us: **powersets**.

For a given set X, the powerset of X, written $\wp(X)$, is

$$\wp(X) \stackrel{\text{def}}{=} \{T \mid T \subseteq X\}$$

$\wp(X)$ is a complete lattice because:

- it comes with a relation $\subseteq$ (set inclusion) that is reflexive, transitive, and antisymmetric.
- it is closed under union and intersection

($\cup$ and $\cap$ give least upper bounds and greatest lower bounds for $\subseteq$)

A **partially ordered set** (or **poset**): a non-empty set with a relation on its elements that is reflexive, transitive, and antisymmetric.

A **complete lattice**: a poset with all joins (least upper bounds) and (hence) also all meets (greatest lower bounds).

Fixed-point Theorem

NB: Complete lattices are “dualisable” structures: reverse the arrows and you get another complete lattice. Similarly, statements on complete lattices can be dualised.

For simplicity, we will focus on complete lattices produced by the powerset construction. But all statements can be generalised to arbitrary complete lattices

An endofunction F is monotone if $T \subseteq T'$ implies $F(T) \subseteq F(T')$

Theorem [Fixed-point Theorem] If $F : \wp(X) \to \wp(X)$ is monotone, then

$$\mathtt{lfp}(F) = \bigcap \{T \mid F(T) \subseteq T\}$$

$$\mathtt{gfp}(F) = \bigcup \{T \mid T \subseteq F(T)\}$$

(the meet of the pre-fixed points, the join of the post-fixed points)

In fact, the theorem tells us more: the set of fixed points is itself a complete lattice, and the same for the sets of pre-fixed points and post-fixed points.

Sets coinductively and inductively defined by F

Given a complete lattice produced by the powerset construction, and an endofunction F on it, the sets:

$$F_{\mathtt{coind}} \stackrel{\text{def}}{=} \bigcup \{x \mid x \subseteq F(x)\}$$

$$F_{\mathtt{ind}} \stackrel{\text{def}}{=} \bigcap \{x \mid F(x) \subseteq x\}$$

are the sets *coinductively defined by* F, and *inductively defined by* F.

Induction and coinduction proof principles

We immeditately derive:

if $x \subseteq F(x)$ then $x \subseteq F_{\mathtt{coind}}$ **(coinduction proof principle)**

if $F(x) \subseteq x$ then $F_{\mathtt{ind}} \subseteq x$ **(induction proof principle)**

By the Fixed-point Theorem: when F monotone, $F_{\mathtt{coind}}$ is the greatest fixed point of F, and dually for $F_{\mathtt{ind}}$.

Hence for monotone functions we can rephrase the principles thus:

Induction and coinduction proof principles, for monotone functions:

if $F(x) \leq x$ then $\mathtt{lfp}(F) \leq x$

if $x \leq F(x)$ then $x \leq \mathtt{gfp}(F)$

We actually know more, from Fixed-point theory.

Eg: in the hypothesis of the Fixed-point Theorem, the set of post-fixed point is a complete lattice.

Thus the join of post-fixed points is itself a post-fixed point, and dually so.

- Inductive definitions give us lfp's (precisely: an inductive definition tells us how to construct the lfp). coinductive definitions give us gfp's.
- On inductively-defined sets the inductive proof principle is the same as the familiar induction technique, and similarly the coinductive proof principles give us the desired coinductive techniques

Definitions by means of rules

Given a set X, a **ground rule on X** is a pair (S, x) with $S \subseteq X$ and $x \in X$

We can write a rule (S, x) as

$$\frac{x_1 \dots x_n \dots}{x}$$

where $\{x_1, \dots, x_n, \dots\} = S$.

A set $\mathcal{R}$ of rules on X yields a monotone endofunction $\Phi_{\mathcal{R}}$, called the **functional of $\mathcal{R}$** (or **rule functional**), on the complete lattice $\wp(X)$, where

$$\Phi_{\mathcal{R}}(T) = \{x \mid (T', x) \in \mathcal{R} \text{ for some } T' \subseteq T\}$$

Exercise Show that $\Phi_{\mathcal{R}}$ above is indeed monotone. Then show that every monotone operator on the complete lattice $\wp(X)$ can be expressed as the functional of some set of rules.

By the Fixed-point Theorem there are least fixed point and greatest fixed point, $\texttt{lfp}(\Phi_{\mathcal{R}})$ and $\texttt{gfp}(\Phi_{\mathcal{R}})$, obtained via the join and meet in the theorem.

They are indeed called the sets **inductively** and **coinductively defined by the rules**.

We also get, from the induction and coinduction principles:

$$\text{if } T \subseteq \Phi_{\mathcal{R}}(T) \text{ then } T \subseteq \texttt{gfp}(\Phi_{\mathcal{R}})$$

$$\text{if } \Phi_{\mathcal{R}}(T) \subseteq T \text{ then } \texttt{lfp}(\Phi_{\mathcal{R}}) \subseteq T$$

Useful to spell out concretely what all this means, beginning with the more familiar induction.

A set T being a pre-fixed point of $\Phi_{\mathcal{R}}$ means that:

for all rules $(S, x) \in \mathcal{R}$,
if $S \subseteq T$, then also $x \in T$.

That is:

(i) the conclusions of each axiom is in T;

(ii) each rule whose premises are in T has also the conclusion in T.

This is precisely the 'forward' closure.

Fixed-point Theory tells us that the the least fixed-point is the least pre-fixed point.

The induction proof principle, then, reads as follows. If you want to prove that all the elements of the set inductively defined by the rules have a property T, then prove that T is a pre-fixed point of $\Phi_{\mathcal{R}}$

This is the familiar way of reasoning inductively on rules. The assumption "$S \subseteq T$" is the **inductive hypothesis**. The base of the induction is given by the axioms of $\mathcal{R}$, where the set S is empty.

Now the case of coinduction. A set T being a post-fixed of $\Phi_{\mathcal{R}}$ means that

for all $t \in T$ there is a rule $(S, t) \in \mathcal{R}$ with $S \subseteq T$

This is precisely the ‘backward’ closure

Thus the greatest fixed point is the greatest set closed backward.

The coinduction proof principle reads thus: That is: if you want to show x is in the set coinductively defined by the rules, then you must find T with $x \in T$ and T post-fixed point of $\Phi_{\mathcal{R}}$

In the literature, the principles in this and previous slide are sometimes referred to as the principles of **rule induction** and of **rule coinduction**.

Exercise Let $\mathcal{R}$ be a set of ground rules, and suppose each rule has a non-empty premise. Show that $\mathtt{lfp}(\Phi_{\mathcal{R}}) = \emptyset$.

Examples finite traces, revised

$$\frac{P \text{ stopped}}{P\downarrow} \qquad \frac{P \xrightarrow{\mu} P' \qquad P'\downarrow}{P\downarrow}$$

As ground rules, these become:

$$\begin{aligned}\mathcal{R}_{\downarrow} \stackrel{\text{def}}{=} \; & \{(\emptyset, P) \mid P \text{ is stopped}\} \\ & \bigcup\{(\{P'\}, P) \mid P \xrightarrow{\mu} P' \text{ for some } \mu\}\end{aligned}$$

This yields the following functional:

$$\Phi_{\mathcal{R}_{\downarrow}}(T) \stackrel{\text{def}}{=} \{P \mid P \text{ is stopped, or there are } P', \mu \text{ with } P' \in T \text{ and } P \xrightarrow{\mu} P'\}$$

The sets ‘closed forward’ are the pre-fixed points of $\Phi_{\mathcal{R}_{\downarrow}}$.

Thus the smallest set closed forward and the associated proof technique become examples of inductively defined set and of induction proof principle.

Examples ω-traces, revised

$$\frac{P \xrightarrow{\mu} P' \qquad P' \restriction_\mu}{P \restriction_\mu}$$

As ground rules, this yields:

$$\mathcal{R}_{\restriction_\mu} \stackrel{\text{def}}{=} \{(\{P'\}, P) \mid P \xrightarrow{\mu} P'\} .$$

This yields the following functional:

$$\Phi_{\mathcal{R}_{\restriction_\mu}}(T) \stackrel{\text{def}}{=} \{P \mid \text{ there is } P' \in T \text{ and } P \xrightarrow{\mu} P'\}$$

Thus the sets 'closed backward' are the post-fixed points of $\Phi_{\mathcal{R}_{\restriction_\mu}}$, and the largest set closed backward is the greatest fixed point of $\Phi_{\mathcal{R}_{\restriction_\mu}}$;

Similarly, the proof technique for ω-traces is derived from the coinduction proof principle.

Example: the λ-calculus

In the case of $\Downarrow$, the rules manipulate pairs of closed λ-terms, thus they act on the set $\Lambda^0 \times \Lambda^0$. The rule functional for $\Downarrow$, written $\Phi_{\Downarrow}$, is

$$\begin{aligned}\Phi_{\Downarrow}(T) \stackrel{\text{def}}{=} & \{(e, e') \mid e = e' = \lambda x.\, e'' \text{ , for some } e'' \} \\ & \bigcup \{(e, e') \mid e = e_1\, e_2 \text{ and} \\ & \qquad \exists\, e_0 \text{ such that } (e_1, \lambda x.\, e_0) \in T \text{ and } (e_0\{^{e_2}/_x\}, e') \in T\} \,.\end{aligned}$$

In the case of $\Uparrow$, the rules are on Λ^0. The rule functional for $\Uparrow$ is

$$\begin{aligned}\Phi_{\Uparrow}(T) \stackrel{\text{def}}{=} & \{e_1\, e_2 \mid e_1 \in T, \} \\ & \bigcup \{e_1\, e_2 \mid e_1 \Downarrow \lambda x.\, e_0 \text{ and } e_0\{^{e_2}/_x\} \in T\}.\end{aligned}$$

Example: the finite lists

Let F be this function (from sets to sets):

$$F(T) \stackrel{\text{def}}{=} \{\mathtt{nil}\} \cup \{\mathtt{cons}(a, s) \mid a \in A, s \in T\}$$

F is monotone, and $\mathrm{finLists} = \mathtt{lfp}(F)$. (i.e., $\mathrm{finLists}$ is the smallest set solution to the equation $\mathcal{L} = \mathtt{nil} + \mathtt{cons}(A, \mathcal{L})$).

From the induction and coinduction principles, we infer: Suppose $T \subseteq \mathrm{finLists}$. If $F(T) \subseteq T$ then $T \subseteq \mathrm{finLists}$ (hence $T = \mathrm{finLists}$).

Proving $F(T) \subseteq T$ requires proving

- $\mathtt{nil} \in T$;
- $\ell \in \mathrm{finLists} \cap T$ implies $\mathtt{cons}(a, \ell) \in T$, for all $a \in A$.

This is the same as the familiar induction technique for lists

Note: F is defined the class of all sets, rather than on a powerset; the class of all sets is not a complete lattice (because of paradoxes such as Russel's), but the constructions that we have seen for lfp and gfp of monotone functions apply.

Example: mathematical induction

The rules are : $\dfrac{}{0}$ $\qquad$ $\dfrac{n}{n+1}$ (for all $n \geq 0$)

We thus obtain the natural numbers as the least fixed point of a rule functional.

This characterisation justifies the common proof principle of induction on the natural numbers, called **mathematical induction**: if a property on the naturals holds at 0 and, whenever it holds at n, it also holds at $n+1$, then the property is true for all naturals.

A variant induction on the natural numbers: the inductive step assumes the property at all numbers less than or equal to n

This corresponds to a variant presentation of the natural numbers, where the rules are:

$$\frac{}{0} \qquad \frac{0, 1, \ldots, n}{n+1} \text{ (for all } n \geq 0)$$

Other examples:

Structural induction, Induction on derivation proofs, Transition induction, Well-founded induction, Transfinite induction

Well-founded induction

Given a well-founded relation $\mathcal{R}$ on a set X, and a property T on X, to show that $X \subseteq T$ (the property T holds at all elements of X), it suffices to prove that, for all $x \in X$: if $y \in T$ for all y with $y \mathcal{R} x$, then also $x \in T$.

mathematical induction, structural induction can be seen as special cases

Well-founded induction is indeed the natural generalisation of mathematical induction to sets and, as such, it is frequent to find it in Mathematics and Computer Science.

Example: proof of a property reasoning on the lexicographical order on pairs of natural numbers

We can derive well-founded induction from fixed-point theory in the same way as we did for rule induction.

In fact, we can reduce well-founded induction to rule induction taking as rules, for each $x \in X$, the pair (S, x) where S is the set $\{y \mid y \mathcal{R} x\}$ and $\mathcal{R}$ the well-founded relation.

Note that the set inductively defined by the rules is precisely X; that is, any set equipped with a well-founded relation is an inductive set.

Transfinite induction

The extension of mathematical induction to ordinals

Transfinite induction says that to prove that a property T on the ordinals holds at all ordinals, it suffices to prove, for all ordinals α: if $\beta \in T$ for all ordinals $\beta < \alpha$ then also $\alpha \in T$.

In proofs, this is usually split into three cases:

(i) $0 \in T$;

(ii) for each ordinal α, if $\alpha \in T$ then also $\alpha + 1 \in T$;

(iii) for each limit ordinal β, if $\alpha \in \mathcal{T}$ for all $\alpha < \beta$ then also $\beta \in T$.

Transfinite induction acts on the ordinals, which form a proper class rather than a set.

As such, we cannot derive it from the fixed-point theory presented.

However, in practice, transfinite induction is used to reason on sets, in cases where mathematical induction is not sufficient because the set has 'too many' elements.

In these cases, in the transfinite induction each ordinal is associated to an element of the set. Then the $<$ relation on the ordinals is a well-founded relation on a set, so that transfinite induction becomes a special case of well-founded induction on sets.

Another possibility: lifting the theory of induction to classes.

Function definitions by recursion and corecursion

One often finds functions defined by means of systems of equations. Such definitions may follow the schema of **recursion** or **corecursion**.

Examples on the well-founded set of the natural numbers:

the factorial function

$$f(0) = 1 \qquad f(n+1) = n \times f(n)$$

An example of structural recursion is the function f that defines the number of λ-abstractions in a λ-term:

$$f(x) = 0 \qquad f(\lambda x.\, e) = 1 + f(e) \qquad f(e\, e') = f(e) + f(e')$$

It is possible to define patterns of equations for well-founded recursion, and prove that whenever the patterns are respected the functions specified exist and are unique. The proof makes use of well-founded induction twice, to prove that such functions exist and to prove its unicity

While a function defined by recursion acts on the elements of an inductive set, one defined by **corecursion** produces an element of a coinductive set.

An equation for a corecursive function specifies the immediate observables of the element returned by the function

for instance, if the element is an infinite list, the equation should tell us specify the head of the list.

Examples are the definitions of the functions `map`, `iterate`

As in the case of recursion, so for corecursion one can produce general equation schemata, and prove that any system of equations satisfying the schemata defines a unique function (or unique functions, in case of mutually recursive equations)

Enhancements of the principles

Theorem **Let F be a monotone endofunction on a complete lattice L, and y a post-fixed point of F (i.e., $y \leq F(y)$). Then**

$$\mathtt{gfp}(F) = \bigcup \{x \mid x \leq F(x \cup y)\}$$

principle of coinduction up-to $\cup$:

> **Let F be a monotone endofunction on a complete lattice, and suppose $y \leq F(y)$;**
> **then $x \leq F(x \cup y)$ implies $x \leq \mathtt{gfp}(F)$.**

Theorem **Let F be a monotone endofunction on a complete lattice L, and $\bullet : L \times L \to L$ an associative function such that:**

1. **for all $x, y, x', y' \in L$, whenever both $x \leq F(x')$ and $y \leq F(y')$, then $x \bullet y \leq F(x' \bullet y')$;**
2. **for all x with $x \leq F(x)$ we have both $x \leq x \bullet \mathtt{gfp}(F)$ and $x \leq \mathtt{gfp}(F) \bullet x$.**

Then

$$\mathtt{gfp}(F) = \bigcup \{x \mid x \leq F(\mathtt{gfp}(F) \bullet x \bullet \mathtt{gfp}(F))\}$$

principle of coinduction up-to gfp:

> **Let F be a monotone endofunction on a complete lattice L, and $\bullet : L \times L \to L$ an associative function for which the assumptions (1) and (2) of Theorem above hold; then $x \leq F(\mathtt{gfp}(F) \bullet x \bullet \mathtt{gfp}(F))$ implies $x \leq \mathtt{gfp}(F)$.**

Equality on coinductive data types

On infinite lists (more generally coinductively defined sets) proving equality may be delicate (they can be “infinite objects”, hence one cannot proceed inductively, eg on their depth)

We can prove equalities adapting the idea of bisimulation.

We show this for $\texttt{FinInfLists}_A$

The coinductive definition tells us what can be observed

We can make this explicit in $\texttt{FinInfLists}_A$ defining an LTS on top of the lists:

$$\langle a \rangle \bullet s \xrightarrow{a} s$$

Lemma 1 For $s, t \in \texttt{FinInfLists}_A$, it holds that $s = t$ if and only if $s \sim t$.

Example

$$\begin{aligned} \texttt{map}\ f\ \texttt{nil} &= \texttt{nil} \\ \texttt{map}\ f\ (\langle a \rangle \bullet s) &= \langle f(a) \rangle \bullet \texttt{map}\ f\ s \end{aligned}$$

$$\texttt{iterate}\ f\ a = \langle a \rangle \bullet \texttt{iterate}\ f\ f(a)$$

Thus `iterate` $f\ a$ builds the infinite list

$$\langle a \rangle \bullet \langle f(a) \rangle \bullet \langle f(f(a)) \rangle \bullet \ldots$$

Show that, for all $a \in A$:

$$\texttt{map}\ f\ (\texttt{iterate}\ f\ a) = \texttt{iterate}\ f\ f(a)$$

Proof

$$\mathcal{R} \stackrel{\text{def}}{=} \{(\langle a \rangle \bullet \texttt{map}\ f\ (\texttt{iterate}\ f\ a), \langle a \rangle \bullet \texttt{iterate}\ f\ f(a)) \mid a \in A\}$$

is a bisimulation

Let $(P, Q) \in \mathcal{R}$, for $P \stackrel{\text{def}}{=} \langle a \rangle \bullet \texttt{map}\ f\ (\texttt{iterate}\ f\ a)$

$Q \stackrel{\text{def}}{=} \langle a \rangle \bullet \texttt{iterate}\ f\ f(a)$

We have

$$P \xrightarrow{a} \texttt{map}\ f\ (\texttt{iterate}\ f\ a) \stackrel{\text{def}}{=} P'$$
$$Q \xrightarrow{a} \texttt{iterate}\ f\ f(a) \stackrel{\text{def}}{=} Q'$$

Applying the definition of `iterate`,

$$Q' = \langle f(a) \rangle \bullet \texttt{iterate}\ f\ f(f(a)) \stackrel{\text{def}}{=} Q''$$

Similarly,

$$P' = \texttt{map}\ f\ \langle a \rangle \bullet (\texttt{iterate}\ f\ f(a))$$

and now, from the definition of `map`,

$$= \langle f(a) \rangle \bullet \texttt{map}\ f\ (\texttt{iterate}\ f\ f(a)) \stackrel{\text{def}}{=} P''$$

We have $(P'', Q'') \in \mathcal{R}$ (as $f(a) \in A$). Moreover, since by Lemma 1 $= \subseteq \sim$, also $P' \sim P''$ and $Q' \sim Q''$.

Summarising, we have showed that $P \xrightarrow{a} P''$ and $Q \xrightarrow{a} Q''$ and $P'' \; \mathcal{R} \; Q''$. This concludes the proof that $\mathcal{R}$ is a bisimulation

Continuity and cocontinuity

Constructive characterisations of least fixed point's and greatest fixed point's are obtained via another important theorem of fixed-point theory

Definition 5 An endofunction on a complete lattice is:

- *continuous* if for all sequences $T_0, T_1 \ldots$ of increasing points in the lattice (i.e., $T_i \subseteq T_{i+1}$, for $i \geq 0$) we have $F(\bigcup_i T_i) = \bigcup_i F(T_i)$.
- *cocontinuous* if for all sequences $T_0, T_1 \ldots$ of decreasing points in the lattice (i.e., $T_{i+1} \subseteq T_i$, for $i \geq 0$) we have $F(\bigcap_i T_i) = \bigcap_i F(T_i)$.

For an endofunction F on a complete lattice, $F^n(x)$ indicates the n-th iteration of F starting from the point x:

$$\begin{array}{rcl} F^0(x) & \stackrel{\text{def}}{=} & x \\ F^{n+1}(x) & \stackrel{\text{def}}{=} & F(F^n(x)) \end{array}$$

Then we set:

$$\begin{array}{rcl} F^{\cap\omega}(x) & \stackrel{\text{def}}{=} & \bigcap_{n\geq 0} F^n(x) \\ F^{\cup\omega}(x) & \stackrel{\text{def}}{=} & \bigcup_{n\geq 0} F^n(x) \end{array}$$

Theorem 1 For a cocontinuous endofunction F on a complete lattice we have:

$$\mathtt{gfp}(F) = F^{\cap\omega}(\top)\,.$$

Dually, if F is continuous:

$$\mathtt{lfp}(F) = F^{\cup\omega}(\bot)$$

where $\top$ and $\bot$ are the top and bottom elements of the lattice

If F is not cocontinuous, and only monotone, we only have $\mathtt{gfp}(F) \leq F^{\cap\omega}(\top)$. The converse need not hold, as the following exercise shows.

Exercise 5 Let L be the set of negative integers plus the elements $-\omega$ and $-(\omega+1)$, with the expected ordering $-n \geq -\omega \geq -(\omega+1)$, for all n. Let now F be the following function on L:

$$\begin{array}{rcl} F(-n) & = & -(n+1) \\ F(-\omega) & = & -(\omega+1) \\ F(-(\omega+1)) & = & -(\omega+1) \end{array}$$

Show that F is monotone but not cocontinuous, and we have $F^{\cap\omega}(\top) = -\omega$ and $\mathtt{gfp}(F) = -(\omega+1)$ □

However, if it happens that $F^{\cap\omega}(\top)$ is a fixed point, then we are sure that it is indeed the greatest fixed point. Having only monotonicity, to reach the greatest fixed point using induction, we need to iterate over the transfinite ordinals. The dual statement, for continuity and least fixed points, also holds.

Theorem 2 Let F be a monotone endofunction on a complete lattice L, and define $F^\lambda(\top)$, where λ is an ordinal, as follows:

$$\begin{array}{lcll} F^0(\top) & \stackrel{\text{def}}{=} & \top & \\ F^\lambda(\top) & \stackrel{\text{def}}{=} & F(\bigcap_{\beta<\lambda} F^\beta(\top)) & \text{for } \lambda > 0 \end{array}$$

Define also $F^\infty(\top) \stackrel{\text{def}}{=} \bigcap_\lambda F^\lambda(\top)$. Then $F^\infty(\top) = \mathtt{gfp}(F)$.

As the ordinals are linearly ordered, and each ordinal is either the successor of another ordinal or the least upper bound of all its predecessors, the above definition can also be given thus:

$$\begin{array}{rcll} F^0(\top) & \stackrel{\text{def}}{=} & \top & \\ F^{\lambda+1}(\top) & \stackrel{\text{def}}{=} & F(F^\lambda(\top)) & \text{for successor ordinals} \\ F^\lambda(\top) & \stackrel{\text{def}}{=} & F(\bigcap_{\beta<\lambda} F^\beta(\top)) & \text{for limit ordinals} \end{array}$$

Thus, on the naturals, the definitions of the F^n used in Theorem 1 coincides with those used in Theorem 2, which explains why the notation is the same.

Continuity and cocontinuity, for rules

The functional given by a set of rules need not be continuous or cocontinuous. As an example, consider a rule

$$\frac{a_1 \quad \dots \quad a_n \quad \dots}{a}$$

and call ϕ is the associated functional. Let $T_n = \{a_1, \dots, a_n\}$. We have $a \in \phi(\bigcup_n T_n)$, but $a \notin \bigcup_n \phi(T_n)$, which proves that ϕ is not continuous. We can recover continuity and cocontinuity for rule functionals adding some conditions.

The duality is less obvious, and needs some care.

Definition 6 A set $\mathcal{R}$ of rules is *finite in the premises*, briefly FP, if for each rule $(S, x) \in \mathcal{R}$ the premise set S is finite. □

Exercise 6 Show that if the set of rules $\mathcal{R}$ is FP, then $\Phi_{\mathcal{R}}$ is continuous; conclude that $\mathtt{lfp}(\Phi_{\mathcal{R}}) = \Phi_{\mathcal{R}}^{\cup\omega}(\emptyset)$. □

the statement of Exercise 6 does not hold for cocontinuity. As a counterexample, take $X = \{b\} \cup \{a_1, \ldots, a_n, \ldots\}$, and the set of rules $(\{a_i\}, b)$, for each i, and let Φ be the corresponding rule functional. Thus $\Phi(T) = \{b\}$ if there is i with $a_i \in T$, otherwise $\Phi(T) = \emptyset$. Consider now the sequence of decreasing sets $T_0, \ldots, T_n, \ldots$, where

$$T_i \stackrel{\text{def}}{=} \{a_j \mid j \geq i\}$$

We have $\Phi(\bigcap_n T_n) = \emptyset$, but $\bigcap_n \Phi(T_n) = \{b\}$.

To obtain cocontinuity we need some finiteness conditions on the conclusions of the rules (rather than on the premises as for continuity).

Definition 7 A set of rules $\mathcal{R}$ is *finite in the conclusions*, briefly FC, if for each x, the set $\{S \mid (S, x) \in \mathcal{R}\}$ is finite (i.e., there is only a finite number of rules whose conclusion is x; note that, by contrast, each premise set S may itself be infinite). □

Theorem 3 If a set of rules $\mathcal{R}$ is FC, then $\Phi_{\mathcal{R}}$ is cocontinuous. □

Exercise 7 Prove Theorem 3. □

Corollary 1 If a set of rules $\mathcal{R}$ on X is FC, then $\mathtt{gfp}(\Phi_{\mathcal{R}}) = \Phi_{\mathcal{R}}^{\bigcap\omega}(X)$. □

Without FC, and therefore without cocontinuity, we have nevertheless $\mathtt{gfp}(\Phi_{\mathcal{R}}) \subseteq \Phi_{\mathcal{R}}^{\bigcap\omega}(X)$.

With the FP or FC hypothesis we are thus able of applying the Continuity/Cocontinuity Theorem 1. For FP and continuity, the theorem tells us that given some rules $\mathcal{R}$, the set inductively defined by $\mathcal{R}$ can be obtained as $\Phi_{\mathcal{R}}^{\bigcup\omega}(\emptyset)$, that is, the limit of the increasing sequence of sets

$$\emptyset, \Phi_{\mathcal{R}}(\emptyset), \Phi_{\mathcal{R}}(\Phi_{\mathcal{R}}(\emptyset)), \Phi_{\mathcal{R}}(\Phi_{\mathcal{R}}(\Phi_{\mathcal{R}}(\emptyset))), \ldots.$$

This means that we construct the inductive set starting with the empty set, adding to it the conclusions of the axioms in $\mathcal{R}$ ($\Phi_{\mathcal{R}}(\emptyset)$), and then repeatedly adding elements following the inference rules in $\mathcal{R}$ in a 'forward' manner. This corresponds to the usual constructive way of interpreting inductively a bunch of rules

As usual, the case for coinductively defined sets is dual.

Bisimulation as a fixed-point

Consider the function $F_\sim : \wp(Pr \times Pr) \to \wp(Pr \times Pr)$ so defined.

$F_\sim(\mathcal{R})$ is the set of all pairs (P, Q) s.t.:

1. $\forall \mu, P'$ s.t. $P \xrightarrow{\mu} P'$, then $\exists Q'$ such that $Q \xrightarrow{\mu} Q'$ and $P' \; \mathcal{R} \; Q'$;
2. $\forall \mu, Q'$ s.t. $Q \xrightarrow{\mu} Q'$, then $\exists P'$ such that $P \xrightarrow{\mu} P'$ and $P' \; \mathcal{R} \; Q'$.

We have:

- $F_\sim$ is monotone;
- $\mathcal{R}$ is a bisimulation iff $\mathcal{R} \subseteq F_\sim(\mathcal{R})$.
- $\sim = \mathtt{gfp}(F_\sim)$;

Stratification of bisimilarity

Continuity, operationally:

Consider the following sequence of equivalences, inductively defined:

$\sim_0 \stackrel{\text{def}}{=} \mathit{Pr} \times \mathit{Pr}$

$P \sim_{n+1} Q \stackrel{\text{def}}{=}$:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim_n Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim_n Q'$.

Then set:

$\sim_\omega \stackrel{\text{def}}{=} \bigcap_n \sim_n$

We have, for all $0 \leq n < \omega$:

$\sim_n = F^n_\sim(\mathit{Pr})$, and $\sim^\omega = F^{\bigcap\omega}_\sim(\mathit{Pr})$

Theorem 4 On processes that are image-finite: $\sim = \sim_\omega$

Image-finite processes :

each reachable state can only perform a finite set of transitions.

Abbreviation: $a^n \stackrel{\text{def}}{=} a. \ldots . a. \mathbf{0}$ (n times)

Example: $\sum_{1 \leq i \leq n} a^n$ (note: n is fixed)

Non-example: $P \stackrel{\text{def}}{=} \sum_{1 \leq i < \omega} a^n$

In the theorem, image-finiteness is necessary:

$P \sim_\omega P + a^\omega$ but $P \not\sim P + a^\omega$

The stratification of bisimilarity given by continuity is also the basis for **algorithms** for mechanically checking bisimilarity and for minimisation of the state-space of a process

These algorithms work on processes that are **finite-state** (ie, each process has only a finite number of possible derivaties)

They proceed by progressively refining a partition of all processes

In the initial partition, all processes are in the same set

Bisimulation: P-complete

[Alvarez, Balcazar, Gabarro, Santha, '91]

With m transitions, n states:

O(m log n) time and O(m + n) space **[Paige, Tarjan, '87]**

Trace equivalence, testing: PSPACE-complete

[Kannelakis, Smolka, '90; Huynh, Tian, 95]

Some simple process operators (from CCS)

Nil: a terminated process, no transitions

Prefixing Action sequentialisation

$$\textsc{Pre}\frac{}{\mu.P \xrightarrow{\mu} P}$$

Parallel composition

$$\textsc{ParL}\ \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

$$\textsc{ParR}\ \frac{P_2 \xrightarrow{\mu} P_2'}{P_1 \mid P_2 \xrightarrow{\mu} P_1 \mid P_2'}$$

$$\textsc{Com}\ \frac{P_1 \xrightarrow{\mu} P_1' \qquad P_2 \xrightarrow{\overline{\mu}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'}$$

As an example, the process $P \stackrel{\text{def}}{=} (a.\mathbf{0} \mid b.\mathbf{0}) \mid \overline{a}.\mathbf{0}$ has the transitions

$$P \xrightarrow{a} (\mathbf{0} \mid b.\mathbf{0}) \mid \overline{a}.\mathbf{0} \qquad P \xrightarrow{\tau} (\mathbf{0} \mid b.\mathbf{0}) \mid \mathbf{0}$$

$$P \xrightarrow{b} (a.\mathbf{0} \mid \mathbf{0}) \mid \overline{a}.\mathbf{0} \qquad P \xrightarrow{\overline{a}} (a.\mathbf{0} \mid b.\mathbf{0}) \mid \mathbf{0}$$

Choice

$$\textsc{SumL}\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'} \qquad \textsc{SumR}\frac{P_2 \xrightarrow{\mu} P_2'}{P_1 + P_2 \xrightarrow{\mu} P_2'}$$

As an example, the process $P \stackrel{\text{def}}{=} (\overline{a}.\, Q_1 \mid a.\, Q_2) + b.\, R$ has the transitions

$$P \xrightarrow{\tau} Q_1 \mid Q_2 \qquad P \xrightarrow{a} \overline{a}.\, Q_1 \mid Q_2$$
$$P \xrightarrow{\overline{a}} Q_1 \mid a.\, Q_2 \qquad P \xrightarrow{b} R$$

Constants (Recursive process definitions)

Each constant K has a behaviour specified by a set of transitions of the form $K \xrightarrow{\mu} P$.

Example: $K \xrightarrow{a} K$

A specification and an implementation of a counter

Take constants $\texttt{Counter}_n$, for $n \geq 0$, with transitions

$$\texttt{Counter}_0 \xrightarrow{\texttt{up}} \texttt{Counter}_1$$

and, for $n > 0$,

$$\texttt{Counter}_n \xrightarrow{\texttt{up}} \texttt{Counter}_{n+1} \qquad \texttt{Counter}_n \xrightarrow{\texttt{down}} \texttt{Counter}_{n-1} \ .$$

The initial state is $\texttt{Counter}_0$

An implementation of the counter in term of a constant $\texttt{C}$ with transition

$$\texttt{C} \xrightarrow{\texttt{up}} \texttt{C} \mid \texttt{down}.\mathbf{0} \ .$$

Proof

$$\mathcal{R} \stackrel{\text{def}}{=} \{(\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0}, \mathtt{Counter}_n) \;\mid\; n \geq 0\}\,,$$

is a bisimulation up-to $\sim$

Take $(\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0}, \mathtt{Counter}_n)$ in $\mathcal{R}$.

Suppose $\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0} \xrightarrow{\mu} P$.

By inspecting the inference rules for parallel composition: μ can only be either $\mathtt{up}$ or $\mathtt{down}$.

$\mu = \mathtt{up}$. the transition from $\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0}$ originates from $\mathtt{C}$, which performs the transition $\mathtt{C} \xrightarrow{\mathtt{up}} \mathtt{C} \mid \mathtt{down}.\,\mathbf{0}$, and $P = \mathtt{C} \mid \Pi_1^{n+1}\, \mathtt{down}.\,\mathbf{0}$.
Process $\mathtt{Counter}_n$ can answer $\mathtt{Counter}_n \xrightarrow{\mathtt{up}} \mathtt{Counter}_{n+1}$. For $P = P'$ and $Q = \mathtt{Counter}_{n+1}$, this closes the diagram.

The pair being inspected: $(\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0}, \mathtt{Counter}_n)$

Action: $\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0} \xrightarrow{\mu} P$

$\mu = \mathtt{down}$. It must be $n > 0$. The action must originate from one of the $\mathtt{down}.\,\mathbf{0}$ components of $\Pi_1^n\, \mathtt{down}.\,\mathbf{0}$, which has made the transition $\mathtt{down}.\,\mathbf{0} \xrightarrow{\mathtt{down}} \mathbf{0}$.

Therefore $P = \mathtt{C} \mid \Pi_1^n\, P_i$, where exactly one P_i is $\mathbf{0}$ and all the others are $\mathtt{down}.\,\mathbf{0}$.

we have: $P \sim \mathtt{C} \mid \Pi_1^{n-1}\, \mathtt{down}.\,\mathbf{0}$.

Process $\mathtt{Counter}_n$ can answer with the transition $\mathtt{Counter}_n \xrightarrow{\mathtt{down}} \mathtt{Counter}_{n-1}$.

This closes the diagram, for $P' \stackrel{\text{def}}{=} \mathtt{C} \mid \Pi_1^{n-1}\, \mathtt{down}.\,\mathbf{0}$ and $Q \stackrel{\text{def}}{=} \mathtt{Counter}_{n-1}$, as $P'\, \mathcal{R}\, Q$.

The case when $\mathtt{Counter}_n$ moves first and $\mathtt{C} \mid \Pi_1^n\, \mathtt{down}.\,\mathbf{0}$ has to answer is similar.

Weak bisimulation

Consider the processes

$$\tau.\overline{a}.\mathbf{0} \qquad \text{and} \qquad \overline{a}.\mathbf{0}$$

They are not strongly bisimilar.

But we do want to regard them as behaviourally equivalent! τ-transitions represent internal activities of processes, which are not visible.

(Analogy in functional languages: $(\lambda x.\, x)3$ and 3 are semantically the same.)

Internal work (τ-transitions) should be ignored in the bisimulation game. Define:

(i) $\Longrightarrow$ as the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$.

(ii) $\stackrel{\mu}{\Longrightarrow}$ as $\Longrightarrow\stackrel{\mu}{\longrightarrow}\Longrightarrow$ (relational composition).

(iii) $\stackrel{\widehat{\mu}}{\Longrightarrow}$ is $\Longrightarrow$ if $\mu = \tau$; it is $\stackrel{\mu}{\Longrightarrow}$ otherwise.

Definition 8 (weak bisimulation, or observation equivalence) A process relation $\mathcal{R}$ is a *weak bisimulation* if $P \mathcal{R} Q$ implies:

1. if $P \stackrel{\mu}{\Longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$;
2. the converse of (1) on the actions from Q.

P and Q are *weakly bisimilar*, written $P \approx Q$, if $P \mathcal{R} Q$ for some weak bisimulation $\mathcal{R}$.

Why did we study strong bisimulation?

- $\sim$ is simpler to work with, and $\sim \subseteq \approx$; (cf: exp. law)
- the theory of $\approx$ is in many aspects similar to that of $\sim$;
- the differences between $\sim$ and $\approx$ correspond to subtle points in the theory of $\approx$

Are the processes $\tau.\mathbf{0} + \tau.\overline{a}.\mathbf{0}$ and $\overline{a}.\mathbf{0}$ weakly bisimilar ?

Examples of non-equivalence:

$$a + b \not\approx a + \tau.\, b \not\approx \tau.\, a + \tau.\, b \not\approx a + b$$

Examples of equivalence:

$$\tau.\, a \approx a \approx a + \tau.\, a$$

$$a.\, (b + \tau.\, c) \approx a.\, (b + \tau.\, c) + a.\, c$$

These are instances of useful algebraic laws, called the τ *laws*:

Lemma 2 1. $P \approx \tau.\, P$;

2. $\tau.\, N + N \approx N$;

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

In the clauses of Definition 8, the use of $\stackrel{\mu}{\Longrightarrow}$ on the challenger side can be heavy.

For instance, take the CCS process $K \stackrel{\circ}{=} \tau.(a \mid K)$; for all n, we have $K \Longrightarrow (a\,|)^n \mid K$, and all these transitions have to be taken into account in the bisimulation game.

The following definition is much simpler to use (the challenger makes a single move):

Definition 9 A process relation $\mathcal{R}$ is a *weak bisimulation* if $P\ \mathcal{R}\ Q$ implies:

1. if $P \stackrel{\mu}{\longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$;
2. the converse of (1) on the actions from Q (ie, the roles of P and Q are inverted).

Proposition 3 The definitions 8 and 9 of weak bisimulation coincide.

Proof A useful exercise. □

Weak bisimulations “up-to”

Definition 10 (weak bisimulation up-to $\sim$) A process relation $\mathcal{R}$ is a *weak bisimulation up-to* $\sim$ if $P \; \mathcal{R} \; Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \sim \mathcal{R} \sim Q'$;
2. the converse of (1) on the actions from Q.

Exercise 8 If $\mathcal{R}$ is a weak bisimulation up-to $\sim$ then $\mathcal{R} \subseteq \approx$.

Definition 11 (weak bisimulation up-to $\approx$) A process relation $\mathcal{R}$ is a *weak bisimulation up-to* $\approx$ if $P \; \mathcal{R} \; Q$ implies:

1. if $P \stackrel{\mu}{\Longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \approx \mathcal{R} \approx Q'$;
2. the converse of (1) on the actions from Q.

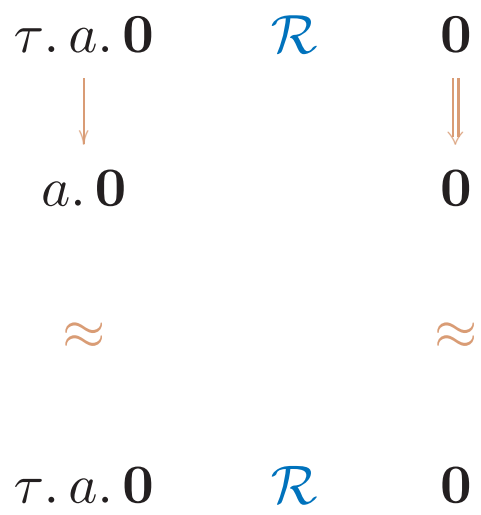
Exercise 9 If $\mathcal{R}$ is a weak bisimulation up-to $\approx$ then $\mathcal{R} \subseteq \approx$.

Enhancements of the bisimulation proof method

- The forms of “up-to” techniques we have seen are examples of **enhancements** of the bisimulation proof method
- Such enhancements are **extremely useful**
  - They are **essential** in π-calculus-like languages, higher-order languages
- **Various forms** of enhancement (“up-to techniques”) exist (up-to context, up-to substitution, etc.)
- They are **subtle**, and not well-understood yet

Example: up-to bisimilarity that fails

In Definition 10 we cannot replace $\sim$ with $\approx$:



Other equivalences

Concurrency theory: models of processes

- LTS
- Petri Nets
- Mazurkiewikz traces
- Event structures
- I/O automata

Process calculi

- CCS [$\longrightarrow \pi$-calculus $\longrightarrow$ Join]
- CSP
- ACP
- Additional features: real-time, probability,...

Behavioural equivalences (and preorders)

- traces
- bisimilarity (in various forms)
- failures and testing
- non-interleaving equivalences (in which parallelism cannot be reduced to non-determinism, cf. the expansion law)
  [causality, location-based]

Depending on the desired level of abstraction or on the tools available, an equivalence may be better than an other.

van Glabbeek, in ’93, listed more than 50 forms of behavioural equivalence, today the listing would be even longer

Rob J. van Glabbeek: The Linear Time - Branching Time Spectrum II, LNCS 715, 1993

Failure equivalence

In CSP equivalence, it is intended that the observations are those obtained from all possible finite experiments with the process

A failure is a pair (μ^+, A), where μ^+ is a trace and A a set of actions. The failure (μ^+, A) belongs to process P if

- $P \xrightarrow{\mu^+} P'$, for some P'
- not $P' \xrightarrow{\tau}$
- not $P' \xrightarrow{a}$, for all $a \in A$

Example: $P \stackrel{\text{def}}{=} a.(b.c.\mathbf{0} + b.d.\mathbf{0})$ has the following failures:

- (ϵ, A) for all A with $a \notin A$.
- (a, A) for all A with $b \notin A$.
- (ab, A) for all A with $\{c, d\} \not\subseteq A$.
- (abc, A) and (abd, A), for all A

Two processes are failure-equivalent if they possess the same failures

Advantages of failure equivalence:

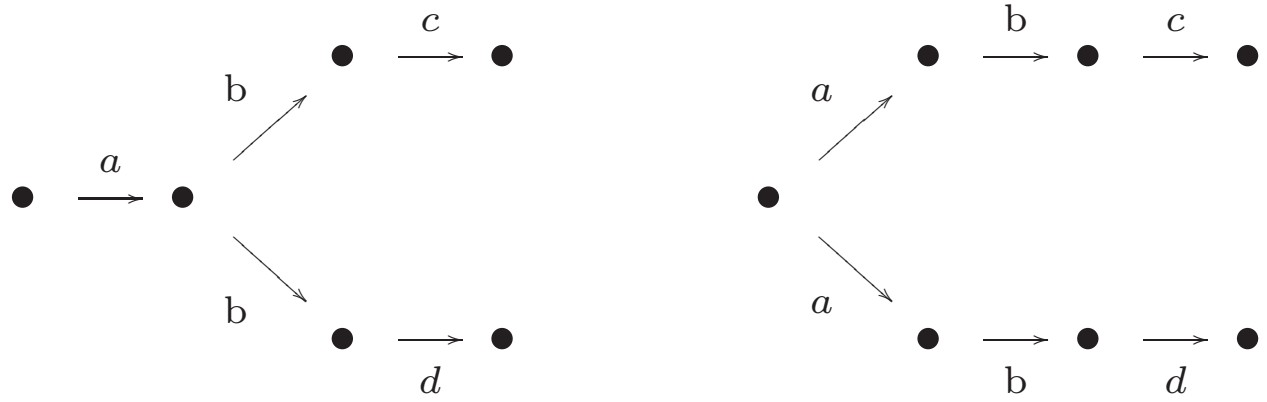
- the coarsest equivalence sensitive to deadlock
- characterisation as testing equivalence

Advantages of bisimilarity:

- the coinductive technique
- the finest reasonable behavioural equivalence for processes
- robust mathematical characterisations

Failure is not preserved, for instance, by certain forms of priority

These processes are failure equivalent but not bisimilar



A law valid for failure but not for bisimilarity:

$$a.\,(b.\,P + b.\,Q) = a.\,b.\,P + a.\,b.\,Q$$

Testing

- The testing theme:

  Processes should be equivalent unless there is some test that can tell them apart
- We first show how to capture bisimilarity this way
- Then we will notice that there are other reasonable ways of defining the language of tests, and these may lead to different semantic notions.
- In this section: processes are (image-finte) LTSs (ie, finitely-branching labelled trees), with labels from a given alphabet of actions `Act`

Bisimulation in a testing scenario

Language for testing:

$$T ::= \mathrm{SUCC} \mid \mathrm{FAIL} \mid a.T \mid \widetilde{a}.T \mid T_1 \wedge T_2 \mid T_1 \vee T_2 \mid \forall T \mid \exists T$$

$(a \in \mathtt{Act})$

The outcomes of an **experiment**, testing a process P with a test T:

$$\mathcal{O}(T, P) \subseteq \{\top, \bot\}$$

$\top$:success

$\bot$: lack of success (failure, or success is never reached)

Notation:

$P\ \mathtt{ref}(a) \stackrel{\text{def}}{=} P$ cannot perform a (ie, there is no P' st $P \xrightarrow{a} P'$)

Outcomes

$$\mathcal{O}(\mathrm{SUCC}, P) = \top$$

$$\mathcal{O}(\mathrm{FAIL}, P) = \bot$$

$$\mathcal{O}(a.T, P) = \begin{cases} \{\bot\} & \text{if } P\ \mathtt{ref}(a) \\ \bigcup\{\mathcal{O}(T, P') \mid P \xrightarrow{a} P'\} & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\widetilde{a}.T, P) = \begin{cases} \{\top\} & \text{if } P\ \mathtt{ref}(a) \\ \bigcup\{\mathcal{O}(T, P') \mid P \xrightarrow{a} P'\} & \text{otherwise} \end{cases}$$

$$\mathcal{O}(T_1 \wedge T_2, P) = \mathcal{O}(T_1, P) \wedge^\star \mathcal{O}(T_1, P)$$

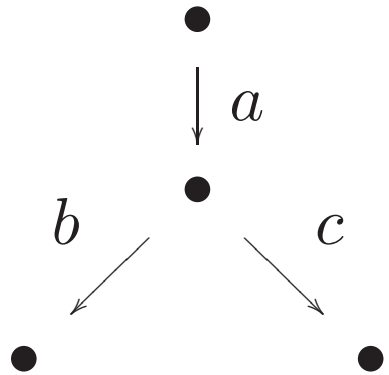
$$\mathcal{O}(T_1 \vee T_2, P) = \mathcal{O}(T_1, P) \vee^\star \mathcal{O}(T_1, P)$$

$$\mathcal{O}(\forall T, P) = \begin{cases} \{\top\} & \text{if } \bot \notin \mathcal{O}(T, P) \\ \{\bot\} & \text{otherwise} \end{cases}$$

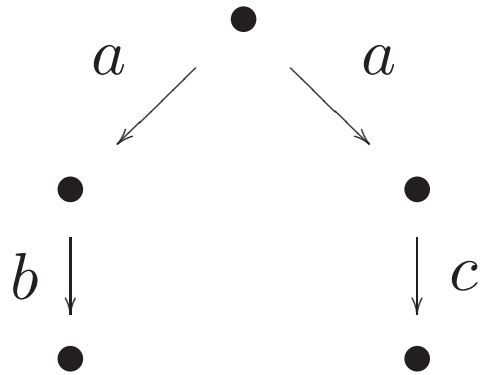
$$\mathcal{O}(\exists T, P) = \begin{cases} \{\top\} & \text{if } \top \in \mathcal{O}(T, P) \\ \{\bot\} & \text{otherwise} \end{cases}$$

where $\wedge^\star$ and $\vee^\star$ are the pointwise extensions of $\wedge$ and $\vee$ to powersets

Examples (a)



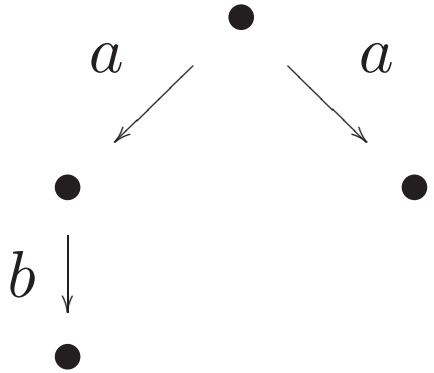
P_1



P_2

For $T_1 = a.\,(b.\,\mathrm{SUCC} \wedge c.\,\mathrm{SUCC})$, we have $\mathcal{O}(T_1, P_1) = \{\top\}$ and $\mathcal{O}(T_1, P_2) = \{\bot\}$

Examples (b)



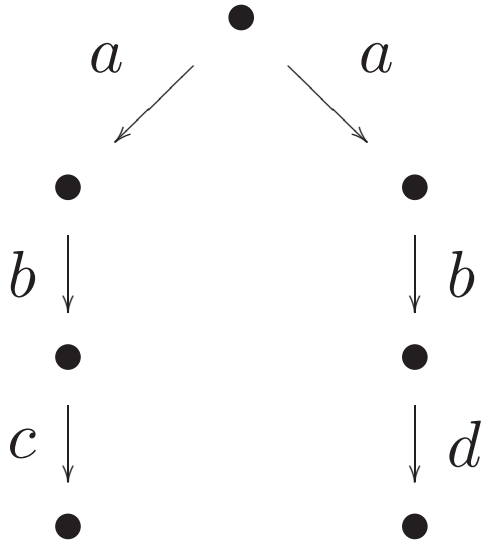
P_3

P_4

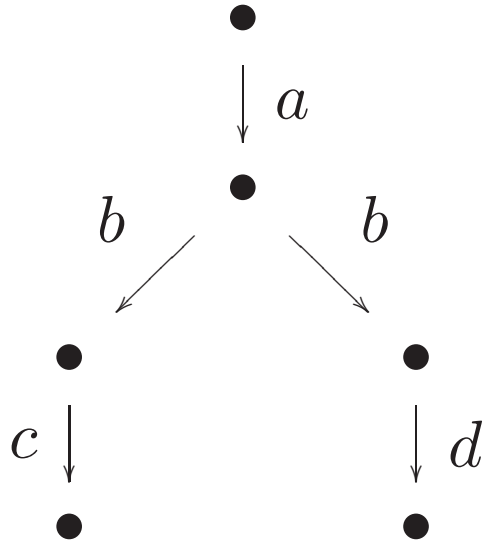
For $T_3 = a.\, b.\, \mathrm{SUCC}$, we have $\mathcal{O}(T_3, P_3) = \{\bot, \top\}$ and $\mathcal{O}(T_3, P_4) = \{\top\}$

For $T_4 = a.\, \widetilde{b}.\, \mathrm{FAIL}$, we have $\mathcal{O}(T_4, P_3) = \{\bot, \top\}$ and $\mathcal{O}(T_4, P_4) = \{\bot\}$

Examples (c)



P_5

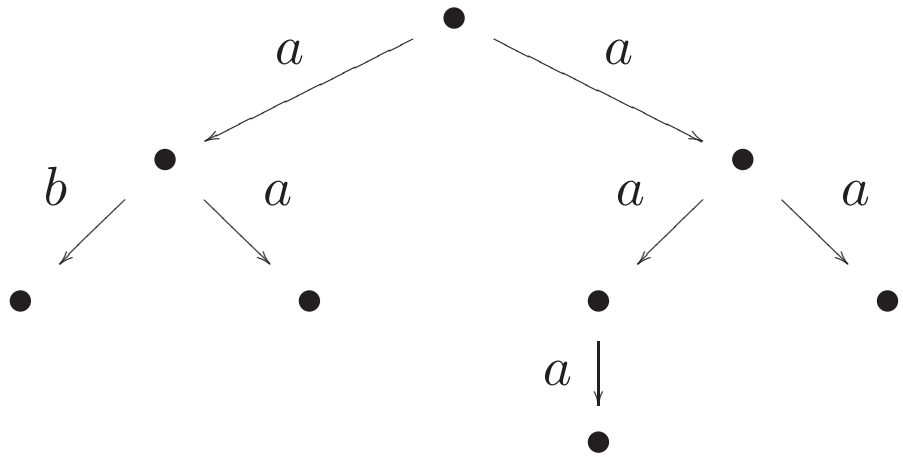


P_6

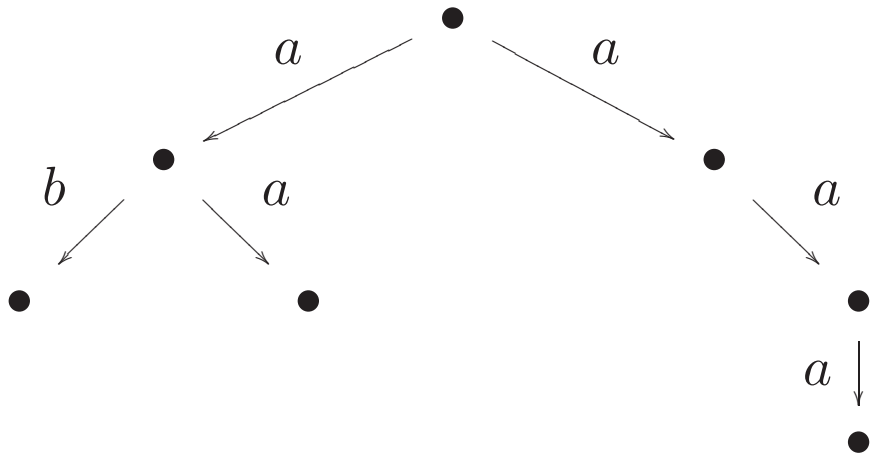
For $T = \exists a.\, \forall b.\, c.\, \mathrm{SUCC}$, we have $\mathcal{O}(T, P_5) = \{\top\}$ and $\mathcal{O}(T, P_6) = \{\bot\}$

Exercise: define other tests that distinguish between P_5 and P_6.

Examples (d)



P_7



P_8

Exercise 10 Define tests that distinguish between P_7 and P_8.

Note: Every test has an inverse:

$$
\begin{aligned}
\overline{\mathrm{SUCC}} &= \mathrm{FAIL} \\
\overline{\mathrm{FAIL}} &= \mathrm{SUCC} \\
\overline{a.\,T} &= \widetilde{a}.\,\overline{T} \\
\overline{\widetilde{a}.\,T} &= a.\,\overline{T} \\
\overline{T_1 \wedge T_2} &= \overline{T_1} \vee \overline{T_2} \\
\overline{T_1 \vee T_2} &= \overline{T_1} \wedge \overline{T_2} \\
\overline{\forall T} &= \exists \overline{T} \\
\overline{\exists T} &= \forall \overline{T}
\end{aligned}
$$

We have:

1. $\bot \in \mathcal{O}(T, P)$ iff $\top \in \mathcal{O}(\overline{T}, P)$
2. $\top \in \mathcal{O}(T, P)$ iff $\bot \in \mathcal{O}(\overline{T}, P)$

The equivalence induced by these tests:

$P \sim_{\mathrm{T}} Q \stackrel{\text{def}}{=}$ for all T, $\mathcal{O}(T, P) = \mathcal{O}(T, Q)$.

Theorem 5 $\sim = \sim_{\mathrm{T}}$

- The proof is along the lines of the proof of characterisation of bisimulation in terms of modal logics (Hennessy-Milner's logics and theorem)
- A similar theorem holds for weak bisimilarity (with internal actions, the definition of the tests may need to be refined)

Testing equivalence

- The previous testing scenario requires considerable control over the processes (eg: the ability to copy their state at any moment)
  One may argue that this is too strong
- An alternative: the tester is a process of the same language as the tested process (in our case: an LTS)
- Performing a test : the two processes attempt to communicate with each other.
- Thus most of the constructs in the previous testing language are no longer appropriate (for instance, because they imply the ability of copying a process)
- To signal success, the tester process uses a special action $w \notin \mathtt{Act}$

Outcomes of running a test

Experiments:

$$E ::= \langle T, P \rangle \mid \top$$

A *run* for a pair $\langle T, P \rangle$: a (finite or infinite) sequence of esperiments E_i such that

1. $E_0 = \langle T, P \rangle$
2. a transition $E_i \xrightarrow{a} E_{i+1}$ is defined by the following rules:

$$\frac{T \xrightarrow{a} T' \qquad P \xrightarrow{a} P'}{\langle T, P \rangle \longrightarrow \langle T', P' \rangle}$$

$$\frac{T \xrightarrow{w} T'}{\langle T, P \rangle \longrightarrow \top}$$

3. the last element of the sequence, say E_k, is such that there is no E' such that $E_k \longrightarrow E'$.

We now set:

$\top \in \mathcal{O}(T, P)$ if $\langle T, P \rangle$ has a run in which $\top$ appears (ie, $\langle T, P \rangle \Longrightarrow \top$)

$\bot \in \mathcal{O}(T, P)$ if there is a run for $\langle T, P \rangle$ in which $\top$ never appears

Testing equivalence ($\simeq$): the equivalence on processes so obtained

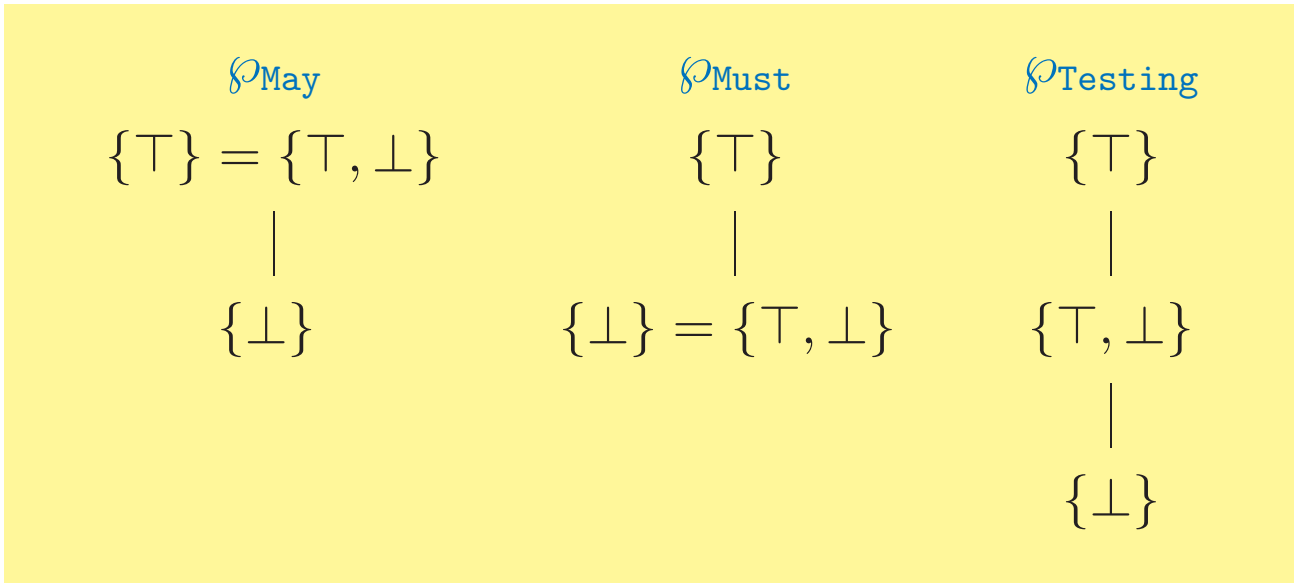
Note: If processes could perform internal actions, then other rules would be needed:

$$\frac{T \stackrel{\tau}{\longrightarrow} T'}{\langle T, P \rangle \longrightarrow \langle T', P \rangle}$$

$$\frac{P \stackrel{\tau}{\longrightarrow} P'}{\langle T, P \rangle \longrightarrow \langle T, P' \rangle}$$

$\mathcal{O}(T, P)$ is a non-empty subset of the 2-point lattice

However, there are 3 ways of lifting such lattice to its non-empty subsets:



$\wp_{\mathtt{May}}$: the possibility of success is essential

$\wp_{\mathtt{Must}}$: failure is disastrous

The resulting equivalences are $\simeq_{\mathtt{May}}$ (**may testing**) and $\simeq_{\mathtt{Must}}$ (**must testing**)

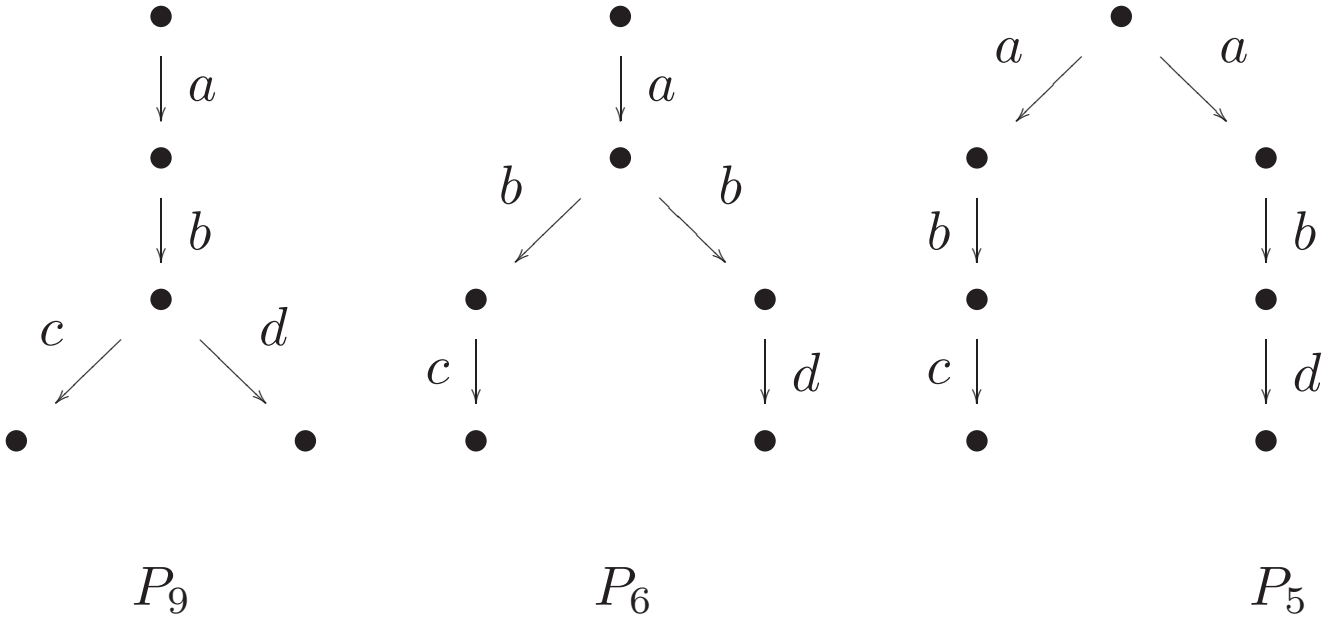
Note: $\simeq_{\mathtt{Testing}}$ is $\simeq$

Theorem 6 1. $\simeq = (\simeq_{\mathtt{May}} \cap \simeq_{\mathtt{Must}})$

2. $\simeq_{\mathtt{May}}$ coincides with trace equivalence

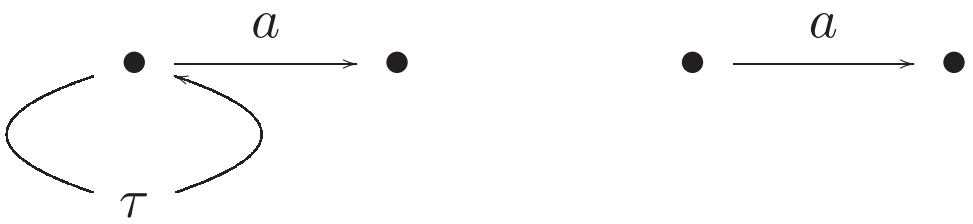
3. $\simeq$ coincides with failure equivalence

Example



P_9 P_6 P_5

P_9	$\simeq_{\mathtt{May}}$	P_5	$\simeq_{\mathtt{May}}$	P_6
P_9	$\not\simeq_{\mathtt{Must}}$	P_5	$\simeq_{\mathtt{Must}}$	P_6
P_9	$\not\simeq$	P_5	$\simeq$	P_6
P_9	$\not\asymp$	P_5	$\not\asymp$	P_6



Q_1 Q_2

In CCS: $Q_1 = \tau^\omega \mid a$, and $Q_2 = a.\mathbf{0}$

Q_1 and Q_2 are **weakly bisimilar**, but **not testing equivalent**

Justification for testing: **bisimulation is insensitive to divergence**

Justification for bisimulation: **testing is not "fair"**

(notions of fair testing have been proposed, and then bisimulation is indeed strictly included in testing)

All equivalences discussed in these lectures reduce parallelism to interleaving, in that

$$a.\mathbf{0} \mid b.\mathbf{0} \text{ is the same as } a.b.\mathbf{0} + b.a.\mathbf{0}$$

Not discussed in these lectures: equivalences that refuse the above equality (called true-concurrency, or non-interleaving)