

Bisimulation and coinduction

Davide Sangiorgi

**Focus Lab.,
INRIA (France) and University of Bologna (Italy)**

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

BASICS 2009, Shanghai

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

These lectures: An introduction to the meaning of behavioural equivalence

We especially discuss bisimulation, as an instance of the coinduction proof method

Outline

- From functions to processes
- Bisimulation
- Induction and coinduction
- Weak bisimulation

From processes to functions

Processes?

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

Concurrent programs 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:

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

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

$$x := 2 \mid x := 2$$

always terminates with $x = 2$. This is not true (why?) for

$$(x := 1; x := x + 1) \mid x := 2$$

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:

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

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 5

$x := 2$ and $x := 1; x := x + 1$

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 **labelled transition system** (LTS) is a triple $(\mathcal{P}, \text{Act}, \mathcal{T})$

where

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

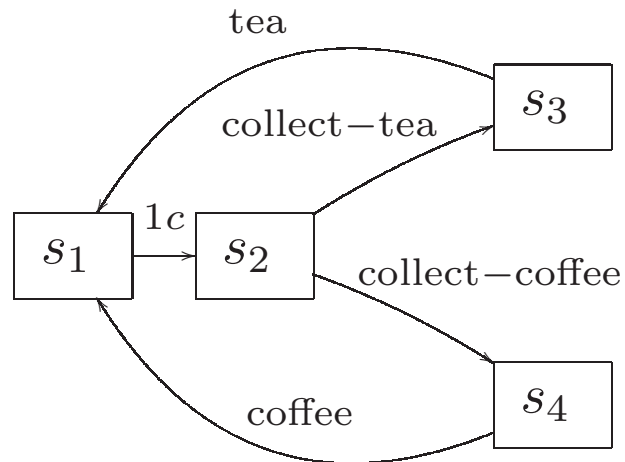
We write $P \xrightarrow{\mu} P'$ if $(P, \mu, P') \in \mathcal{T}$. Meaning: process P accepts an interaction with the environment where P performs action μ and then becomes process P' .

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

Example

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

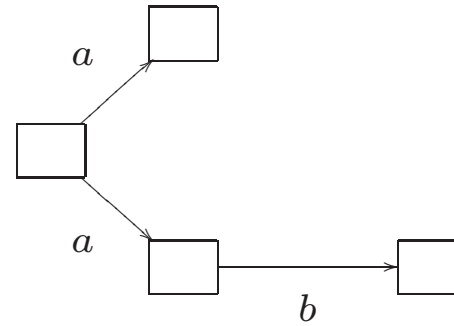
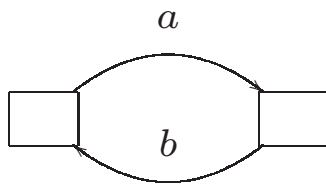
The behaviour of the machine is what we can observe, by interacting with the machine. We can represent such a behaviour as an LTS:



(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.

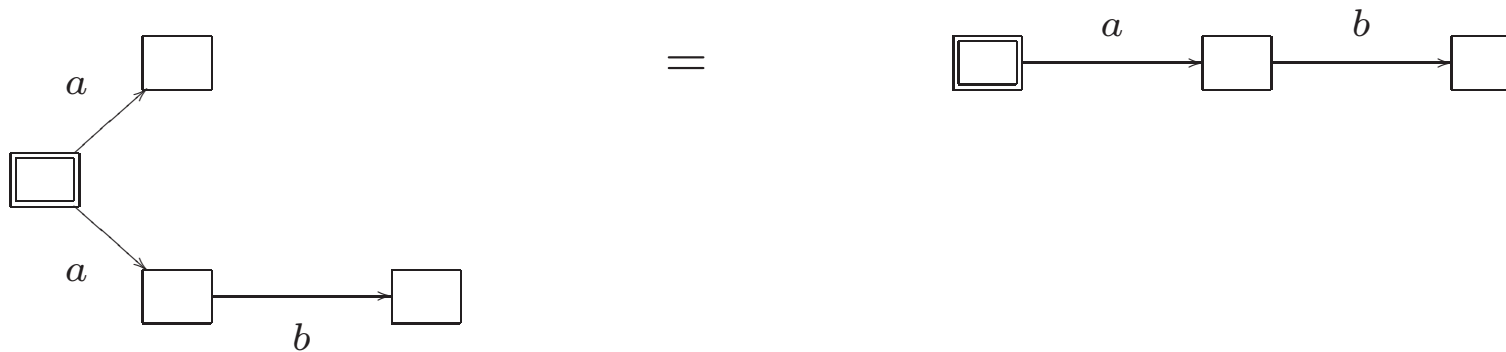
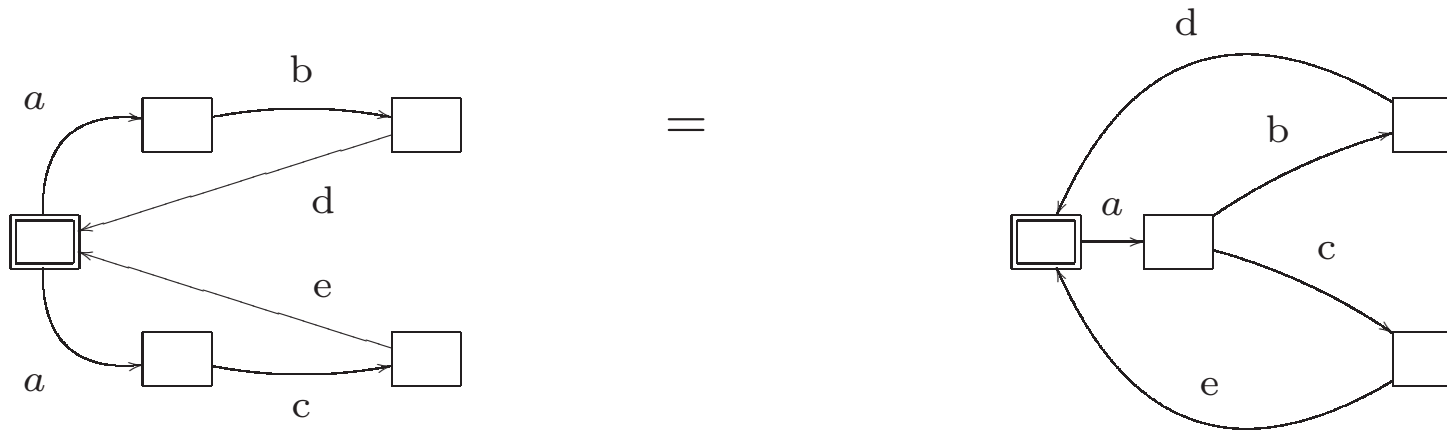
Example (where  indicates the processes we are interested in):



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

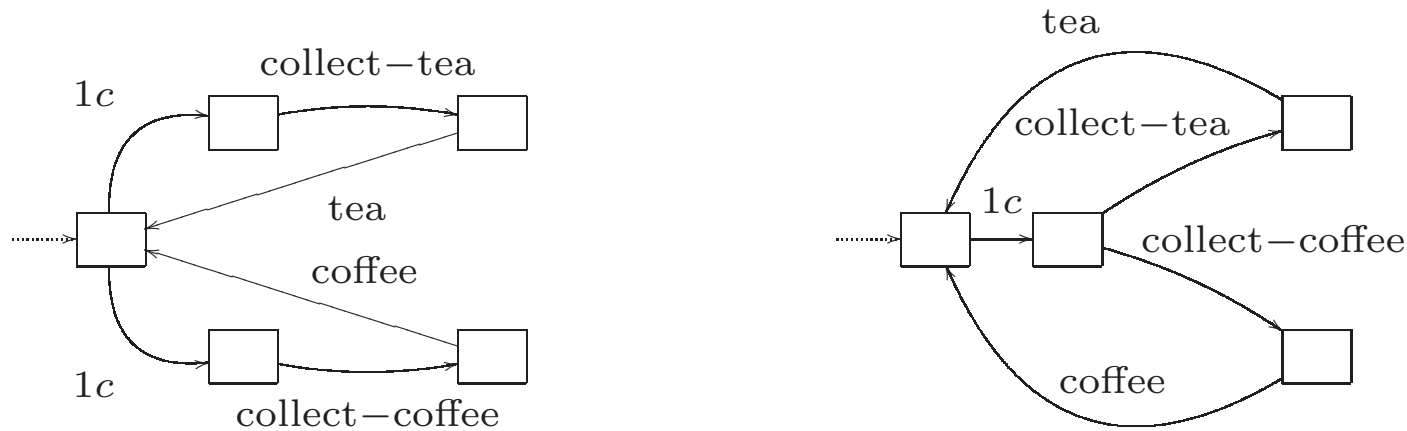
A natural alternative (from automata theory): **trace equivalence**.

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 do 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 3 Prove that the processes at page 12 are bisimilar. Are the processes at page 13 bisimilar?

Proposition 4 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 4(2) suggests an alternative definition of \sim :

Proposition 5 \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 6 Prove Propositions 4-5
(for 4(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 7 (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 8 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 9 (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* 10 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 13.)

- 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:

$$\begin{array}{ccc} \text{A relation } \mathcal{R} \text{ s.t.} & P & \mathcal{R} & Q \\ & \alpha \downarrow & & \downarrow \alpha \\ & P' & \mathcal{R} & Q' \end{array}$$

Bisimilarity (\sim):

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

Hence:

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

(bisimulation proof method)

- The definition of \sim seems circular

(From Proposition 5)

\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?

An example of an inductive definition: finite lists over a set A

$$\frac{}{\text{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \quad a \in A}{\text{cons}(a, \ell) \in \mathcal{L}}$$

Finite lists: the set generated by these rules; i.e., the smallest set closed forward under these rules.

A set T is closed forward if:

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

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

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

An example of an coinductive definition: finite and infinite lists over a set A

$$\frac{}{\text{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \quad a \in A}{\text{cons}(a, \ell) \in \mathcal{L}}$$

Finite and infinite lists: the largest set closed backward under these rules.

T is closed backward if $\forall t \in T$, either $t = \text{nil}$ or $t = \text{cons}(a, \ell)$ for some $\ell \in T$

Constructively:

X = the set of all (finite and infinite) strings with elements from the alphabet $A \cup \{\text{nil}, \text{cons}, (,)\}$

start from X (all strings) and keep removing strings, following the backward-closure, until no more strings can be removed

NB: one can avoid introducing X using *non-well-founded sets* and coalgebras.

An example of an inductive definition: finite traces

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

P has 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:

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

\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 $P \xrightarrow{\mu} P'$ 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 $P \downarrow_{\mu}$, if it is possible to observe an infinite sequence of μ -transitions starting from P .

\downarrow_{μ} has a natural **coinductive** definition in terms of rules:

$$\frac{P \xrightarrow{\mu} P' \quad P' \downarrow_{\mu}}{P \downarrow_{\mu}}$$

\downarrow_{μ} 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 $P \xrightarrow{\mu} P'$.

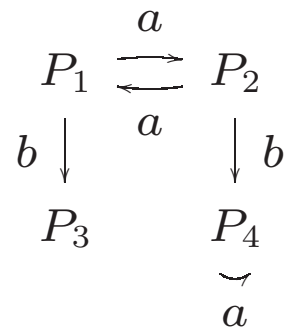
Constructively: you can start from Pr (all processes) and keep removing processes, following the backward-closure, until no more processes can be removed

Hence: to prove that a given process P has an ω -trace under μ it suffices to find some $T \subseteq Pr$ 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 \downarrow_a , hence $P_1 \downarrow_a$ and $P_2 \downarrow_a$ 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 \downarrow_a hold.

An example of an inductive definition: reduction to a value in the λ -calculus

The set Λ of λ -terms (an inductive def!)

$$e ::= x \mid \lambda x. e \mid e_1(e_2)$$

Consider the definition of \Downarrow_n in λ -calculus (convergence to a value):

$$\frac{}{\lambda x. e \Downarrow_n \lambda x. e} \quad \frac{e_1 \Downarrow_n \lambda x. e_0 \quad e_0\{e_2/x\} \Downarrow_n e'}{e_1(e_2) \Downarrow_n e'}$$

\Downarrow_n is the *smallest* relation on λ -terms that is closed forwards under these rules; i.e., the smallest subset C of $\Lambda \times \Lambda$ 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 $e_1(e_2) C e'$.

An example of a coinductive definition: divergence in the λ -calculus

Consider the definition of \uparrow^n (divergence) in λ -calculus :

$$\frac{e_1 \uparrow^n}{e_1(e_2) \uparrow^n} \quad \frac{e_1 \Downarrow_n \lambda x. e_0 \quad e_0\{e_2/x\} \uparrow^n}{e_1(e_2) \uparrow^n}$$

\uparrow^n is the *largest* predicate on λ -terms that is closed backwards under these rules; i.e., the largest subset D of Λ s.t. if $e \in D$ then

- either $e = e_1(e_2)$ and $e_1 \in D$,
- or $e = e_1(e_2)$, $e_1 \Downarrow_n \lambda x. e_0$ and $e_0\{e_2/x\} \in D$.

Hence: to prove e is divergent it suffices to find $E \subseteq \Lambda$ that is closed backwards and with $e \in E$ (coinduction proof technique).

What is the smallest predicate closed backwards?

- * An inductive definition tells us what are the *constructors* for generating all the elements (cf: closure forwards).
- * A coinductive definition tells us what are the *destructors* for decomposing the elements (cf: closure backwards).
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).
- When a definition is give by means of some 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 duality

constructors	destructors
inductive defs	coinductive defs
induction technique	coinductive technique
congruence	bisimulation
least fixed-points	greatest fixed-points

(The dual of a *bisimulation* is a *congruence*: intuitively: a bisimulation is a relation “closed backwards”, a congruence is “closed forwards”)

- In what sense are \sim , \uparrow_μ , \downarrow , etc. fixed-points?
- What is the coinduction proof technique?
- In what sense is coinduction dual to the familiar induction technique?

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

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.)

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

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

$\wp(S)$ is a *complete lattice*.

A *complete lattice* is 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.

We will only consider complete lattices produced by the powerset construction. (Thus: F monotone if $S \subseteq T$ implies $F(S) \subseteq F(T)$)

Theorem 11 (Fixed-point Theorem, or Knaster-Tarski Theorem) If $\mathcal{F} : \wp(X) \rightarrow \wp(X)$ is monotone, then

$$\text{lfp}(\mathcal{F}) = \bigcap \{S \mid \mathcal{F}(S) \subseteq S\}$$

$$\text{gfp}(\mathcal{F}) = \bigcup \{S \mid S \subseteq \mathcal{F}(S)\}$$

(the meet of the pre-fixed points, the join of the 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:

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

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

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

Hence:

$$\text{if } x \subseteq F(x) \text{ then } x \subseteq F_{\text{coind}} \quad (\text{coinduction proof principle}) \quad (1)$$

$$\text{if } F(x) \subseteq x \text{ then } F_{\text{ind}} \subseteq x \quad (\text{induction proof principle}) \quad (2)$$

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

(infact we know more: eg, 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 (1) is the same as the familiar induction technique (cf: example of lists). (2) gives us the coinductive proof technique.

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

$$\Phi_{\mathcal{R}}(S) = \{x \mid (S', x) \in \mathcal{R} \text{ for some } S' \subseteq S\}$$

Exercise 12 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, $\text{lfp}(\Phi_{\mathcal{R}})$ and $\text{gfp}(\Phi_{\mathcal{R}})$, 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 (1) and (2), coinduction and induction proof principles, respectively stating:

$$\text{if } S \subseteq \Phi_{\mathcal{R}}(S) \text{ then } S \subseteq \text{gfp}(\Phi_{\mathcal{R}})$$
$$\text{if } \Phi_{\mathcal{R}}(S) \subseteq S \text{ then } \text{lfp}(\Phi_{\mathcal{R}}) \subseteq S$$

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 $\Phi_{\mathcal{R}}$, that is, (??).

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

Example: finite and ω -traces

In the case of \downarrow , the set of rules is:

$$\mathcal{R}_{\downarrow} \stackrel{\text{def}}{=} \{(\emptyset, P) \mid P \text{ is stopped}\} \\ \cup \{(\{P'\}, P) \mid P \xrightarrow{\mu} P' \text{ for some } \mu\}$$

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.

In the case of \uparrow_μ , the the set of rules is:

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

This yields the following functional:

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

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

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 Φ_{\Downarrow} , is

$$\Phi_{\Downarrow}(T) \stackrel{\text{def}}{=} \{(e, e') \mid e = e' = \lambda x. e'', \text{ for some } e'' \} \\ \cup \{(e, e') \mid e = e_1 e_2 \text{ and} \\ \exists e_0 \text{ such that } (e_1, \lambda x. e_0) \in T \text{ and } (e_0\{e_2/x\}, e') \in T\} .$$

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

$$\Phi_{\Uparrow}(T) \stackrel{\text{def}}{=} \{e_1 e_2 \mid e_1 \in T, \} \\ \cup \{e_1 e_2 \mid e_1 \Downarrow \lambda x. e_0 \text{ and } e_0\{e_2/x\} \in T\} .$$

Example: the finite lists

Let \mathcal{F} be this function (from sets to sets):

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

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

From (1), we infer: Suppose $T \subseteq \text{finLists}$. If $\mathcal{F}(T) \subseteq T$ then $T \subseteq \text{finLists}$ (hence $T = \text{finLists}$).

Proving $\mathcal{F}(T) \subseteq T$ requires proving

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

This is the same as the familiar induction technique for lists

Note: \mathcal{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 : $\bar{0} \quad \frac{n}{n+1}$ (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:

$$\bar{0} \quad \frac{0, 1, \dots, n}{n+1} \text{ (for all } n \geq 0)$$

Bisimulation as a fixed-point

Consider the function $F_{\sim} : \wp(\text{Pr} \times \text{Pr}) \rightarrow \wp(\text{Pr} \times \text{Pr})$ 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;
- $\sim = \text{gfp}(F_{\sim})$;
- \mathcal{R} is a bisimulation iff $\mathcal{R} \subseteq F_{\sim}(\mathcal{R})$.

Continuity

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

Definition 14 An endofunction on a complete lattice is:

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

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

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

Then we set:

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

Theorem 15 For a cocontinuous endofunction \mathcal{F} on a complete lattice we have:

$$\text{gfp}(\mathcal{F}) = F^{\cap\omega}(Pr).$$

Dually, if \mathcal{F} is continuous:

$$\text{lfp}(\mathcal{F}) = F^{\cup\omega}(\perp)$$

A caveat: the function F_{\sim} of which bisimilarity is the gfp may not be cocontinuous! (This is usually the case for *weak* bisimilarity, that we shall introduce later.)

It is cocontinuous if the LTS is finite-branching, meaning that for all P the set $\{P' \mid P \xrightarrow{\mu} P', \text{ for some } \mu\}$ is finite.

On a finite branching LTS, it is indeed the case that

$$\sim = \bigcap_n F_{\sim}^n(Pr \times Pr)$$

where Pr is the set of all processes.

Stratification of bisimilarity

Continuity, operationally:

Consider the following sequence of equivalences, inductively defined:

$$\sim_0 \stackrel{\text{def}}{=} \mathcal{P} \times \mathcal{P}$$

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

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 $0 \leq n < \omega$: $\sim_n = F_{\sim}^n(\text{Pr})$, and $\sim^\omega = F_{\sim}^{\bigcap \omega}(\text{Pr})$

Theorem 16 On processes that are image-finite: $\sim = \sim_\omega$

Image-finite processes :

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

Abbreviation: $a^n \stackrel{\text{def}}{=} a \dots a \cdot \mathbf{0}$ (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 derivatives)

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]

Weak bisimulation

Consider the processes

$$\tau.\bar{a}.0 \quad \text{and} \quad \bar{a}.0$$

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 $\xrightarrow{\tau}$.
- (ii) $\xRightarrow{\mu}$ as $\Longrightarrow \xrightarrow{\mu} \Longrightarrow$ (relational composition).
- (iii) $\xRightarrow{\hat{\mu}}$ is \Longrightarrow if $\mu = \tau$; it is $\xRightarrow{\mu}$ otherwise.

Definition 17 (weak bisimulation, or observation equivalence) A process relation \mathcal{R} is a *weak bisimulation* if PRQ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' s.t. $Q \xrightarrow{\hat{\mu}} Q'$ 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 $\tau. \mathbf{0} + \tau. \bar{a}. \mathbf{0}$ and $\bar{a}. \mathbf{0}$ 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 18 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 17, the use of $\xRightarrow{\mu}$ on the challenger side can be heavy.

For instance, take the CCS process $K \doteq \tau. (a \mid K)$; for all n , we have $K \xRightarrow{} (a \mid)^n \mid K$, 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 19 A process relation \mathcal{R} is a *weak bisimulation* if PRQ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' s.t. $Q \xRightarrow{\hat{\mu}} Q'$ 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 20 The definitions 17 and 19 of weak bisimulation coincide.

Proof A useful exercise. □

Weak bisimulations “up-to”

Definition 21 (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. $Q \xrightarrow{\hat{\mu}} Q'$ and $P' \sim \mathcal{R} \sim Q'$;
2. the converse of (1) on the actions from Q .

Exercise 22 If \mathcal{R} is a weak bisimulation up-to \sim then $\mathcal{R} \subseteq \approx$.

Definition 23 (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 $P \xRightarrow{\mu} P'$, then there is Q' s.t. $Q \xRightarrow{\hat{\mu}} Q'$ and $P' \approx \mathcal{R} \approx Q'$;
2. the converse of (1) on the actions from Q .

Exercise 24 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 21 we cannot replace \sim with \approx :

$$\begin{array}{ccc} \tau . a . \mathbf{0} & \mathcal{R} & \mathbf{0} \\ \downarrow & & \Downarrow \\ a . \mathbf{0} & & \mathbf{0} \\ \approx & & \approx \\ \tau . a . \mathbf{0} & \mathcal{R} & \mathbf{0} \end{array}$$

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]

References

This course is based on the draft book:

- Davide Sangiorgi, *An introduction to bisimulation and coinduction*, Draft, 2009

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