An introduction to bisimulation and coinduction

Davide Sangiorgi

April 9, 2008

We introduce bisimulation and coinduction roughly following the way that led to their discovery in Computer Science. Thus the general topic is the semantics of concurrent languages (or systems), in which several activities, the *processes*, may run concurrently. Central questions are: what is, mathematically, a process? And what does it mean that two processes are "equal"? We seek notions of process and process equality that are both mathematically and practically interesting. For instance, the notions should be amenable to effective techniques for proving equalities, and the equalities themselves should be justifiable, according to the way processes are used.

We hope that the reader will find this way of proceeding helpful to understand the meaning of bisimulation and coinduction. Further, concurrency remains today probably the main application area for bisimulation and coinduction, and plays a central role in the book.

0.1 From functions to processes

If we begin investigating the semantics of concurrent languages, it is natural to check first whether we can adapt to these languages the concepts and techniques that are available for the *sequential* languages, i.e., the languages without constructs for concurrency. This is indeed what researchers did in the 70s, as the work on the semantics of sequential languages had already produced significant results, notably with Scott and Stratchey's *denotational semantics*. In sequential languages, a program is interpreted as a function, which transforms inputs into outputs. This idea is clear in the case of functional languages such as the λ-calculus, but it can also be applied to imperative languages, viewing a program as a function that transforms an initial store (i.e., a memory state) into a final store.

The interpretation of programs as functions, however, in general is unsatisfactory in concurrency. Take, as an example, the following two program fragments in an imperative language:

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

They yield the same function from stores to stores, namely the function that leaves the store unchanged, except of the variable $\mathtt{X}$ whose final value must be 2. Therefore, in this view of programs-as-functions, the two fragments above are "the same" and should be considered equal.

However, the above equality is troublesome if the language to which the two fragments belong is concurrent. For instance, suppose the language has a construct for parallelism, say $P \mid Q$, which, intuitively, allows the parallel execution of the two program arguments P and Q (this rough intuition is sufficient for

the example). Then we may want to try running each fragment together with another fragment such as $\mathtt{X} := 2$. Formally, one says that the two fragments are used in the context

$$[\cdot] \mid \mathtt{X} := 2$$

to fill the hole $[\cdot]$. Now, if we place in the hole the first fragment, $\mathtt{X} := 2$, we get

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

which always terminates with $\mathtt{X} = 2$. This is not true, however, when the hole is filled with the second fragment, $\mathtt{X} := 1; \mathtt{X} := \mathtt{X} + 1$, resulting in

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

as now the final value of $\mathtt{X}$ can be different from 2. For instance, the final value can be 3 if the statement $\mathtt{X} := 2$ is executed after $\mathtt{X} := 1$ but before $\mathtt{X} := \mathtt{X} + 1$.

The example shows that by viewing programs as functions we obtain a notion of program equality that is not preserved by parallel composition: equal arguments to the parallel constructs can produce results that are not anymore equal. Formally, one says that the semantic is not *compositional*, or that the equality on programs is not a *congruence*.

A semantics of a language that is not compositional would not allow us to exploit the structure of the language when reasoning. We cannot for instance, use properties of program components to infer properties of larger systems, or optimise a program component replacing it with an equal but simpler component, as the meaning of the whole program might change.

Another reason why viewing a concurrent program as a function is not appropriate is that 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, for instance in the λ-calculus, programs that do not terminate are undesirable; they are "wrong"; they are perhaps caused by a loop in which the termination condition is wrong. Mathematically, they represent functions that are undefined—hence meaningless—on some arguments.

Also, the behaviour of a concurrent program can be non-deterministic, as shown in the example

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

above. In sequential languages, operators for non-determinism, such as choice, can be dealt with using powersets and powerdomains. For instance, in the λ-calculus, the term $\lambda x.\,(x \oplus x + 1)$, where $\oplus$ indicates the choice construct, could be interpreted as the function that receives an integer x and returns an

element from the set $\{x, x+1\}$. This approach may work (and anyhow is more complicated) for pure non-determinism, but not for the parallelism resulting from the parallel execution of activities, as we have seen above.

If parallel programs are not functions, what are they? They are *processes*. But what is a process? When are two processes equal? These are very fundamental questions for a model of processes. They are also hard questions, that are at the heart of the research in concurrency theory. We shall approach these questions from a simple case, in which the interactions of the process with its environment are particularly simple: they are just synchronisations, without exchange of values. Without the presumption of giving single and definitive answers, we shall strive to isolate the essential concepts.

0.2 Interaction and behaviour

In the example of Section 0.1, the program fragments

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

should be distinguished because they *interact* in a different way with the memory. The difference is harmless within a sequential language, as only the initial and final states are visible to the rest of the world. But if other concurrent entities have access to the same memory locations, then the patterns of the interactions with the memory become significant because they may affect other activities.

This brings up a key word: *interaction*. In concurrency, computation is interaction. Examples are: an access to a memory cell, a query to a data base, the selection of a programme in a washing machine. The participants of an interaction are the *processes* (for instance, in the case of the washing machine, the machine itself and the person selecting the programme are the involved processes). The *behaviour* of a process should tell us *when* and *how* the process can interact with the outside world—its *environment*. Therefore we first need suitable means for representing the behaviour of a process.

0.3 Labelled Transition Systems

As another example of interactions, we consider a vending machine capable of dispensing tea or coffee for 1 coin ($1c$). The machine has a slot for inserting coins, a button for requesting coffee, another button for requests of tea, and an opening for collecting the beverage delivered. The behaviour of the machine is what we can observe, by interacting with the machine. This means experimenting with the machine: pressing buttons and seeing what happens. We can observe which

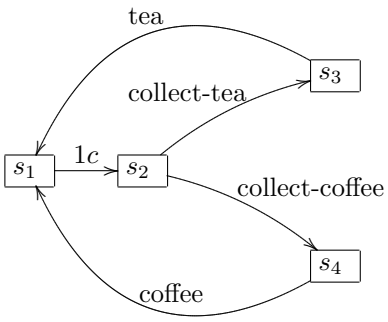


Fig. 0.1. The LTS of a vending machine

buttons go down and when, which beverages we can get and when. Everything else, such as the colour or the shape of the machine, is irrelevant. We are interested in what the machine does, not in what it looks like. We can represent what is relevant of the behaviour of the machine as a *Labelled Transition System* (LTS), as shown, graphically, in Figure 0.1.

An LTS tells us what the states in which a system can be and, for each state, the interactions that are possible from that state. An interaction is represented by a labelled arc; in the LTS terminology, it is called a *transition*. In the case of the vending machine of Figure 0.1, there are 4 states. Initially, the machine is in state s_1. The arc labelled $1c$ between s_1 and s_2 indicates that on state s_1 the machine accepts a coin and, in doing so, it evolves into the state s_2; in s_2 two further transitions are possible, one representing the request for coffee, the other the request for tea; and so on. Other examples of LTSs, in graphical form, are given in Figures 0.2-0.5.

LTSs are the most common structures used to represent the interactions that a system can produce. They are essentially labelled directed graphs. Variant structures are relational structures (i.e., unlabeled directed graphs) and Kripke structures, and it is easy to adapt the concepts we will introduce (notably bisimulation) to them.

Definition 0.3.1 (Labelled Transition Systems) A *Labelled Transition System* (LTS) is a triple $(Pr, Act, \longrightarrow)$ where Pr is a non-empty set called the *domain* of the LTS, Act is the set of *labels*, and $\longrightarrow \subseteq \wp(Cons \times Act \times Pr)$ is the *transition relation*. □

What is "Cons" ?

In the definition above, the elements of Pr are called *states* or *processes*. We will usually call them processes as this is the standard terminology in concur-

rency. We use P, Q to range over such elements, and μ to range over the labels in *Act*. We write $P \xrightarrow{\mu} Q$ when $(P, \mu, Q) \in \longrightarrow$; in this case we call Q a *μ-derivative of P*, or sometimes simply a *derivative of P*. A transition $P \xrightarrow{\mu} Q$ indicates that process P accepts an interaction with the environment, in which P performs action μ and then becomes process Q. If there are $P_1, \ldots, P_n, \mu_1, \ldots, \mu_n$ s.t. $P \xrightarrow{\mu_1} P_1 \ldots P_{n_1} \xrightarrow{\mu_n} P_n$, then P_n is a *derivative* of P under $\mu_1, \ldots, \mu_n$, or simply a *multi-step derivative* of P. In the remainder, we usually do not explicitely indicate the LTS for the processes we write.

0.4 Equality of behaviours

An LTS tells us what is the behaviour of processes. The next question now is: when should two behaviours be considered equal? I.e., what does it mean that two processes are equivalent? Intuitively, two processes should be equivalent if they cannot be distinguished by interacting with them. In the following sections we try to formalise this—very vague—statement.

0.4.1 Isomorphism and trace equivalence

0.4.1.1 Equality in Graph Theory

We have observed that LTSs resemble graphs. We could therefore draw inspiration for our notion of behavioural equality from graph theory. The standard equality on graphs is graph isomorphism. (In mathematics, two structures are isomorphic is a bijection can be established on their components; on graphs the components are the states and the transitions.) Is this notion satisfactory for us?

Certainly, if two LTSs are isomorphic then we expect that the corresponding states give rise to the same interactions and should indeed be regarded as equal. What about the converse, however? Consider the LTSs in Figure 0.2, and the interactions that are possible from the initial processes s_1 and t_1. Both processes just allow us to repeat the sequence of interactions a, b, ad infinitum. It is undeniable that the two processes cannot be distinguished by interactions. However, there is no isomorphisms on the two LTSs, as they have quite different shapes. We have to conclude that graph isomorphism is too strong as a behavioural equivalence for processes.

0.4.1.2 Equality in Automata Theory

LTSs also remind us of something very important in Computer Science: *automata*. The main difference between automata (precisely, we are thinking of non-deterministic automata here) and LTSs is that an automaton has also a

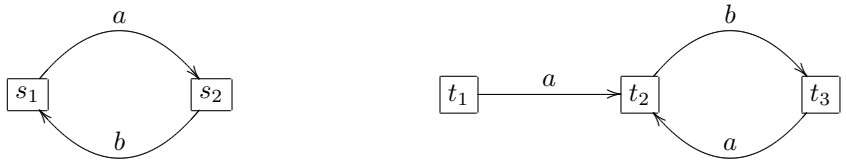


Fig. 0.2. Non-isomorphic LTSs

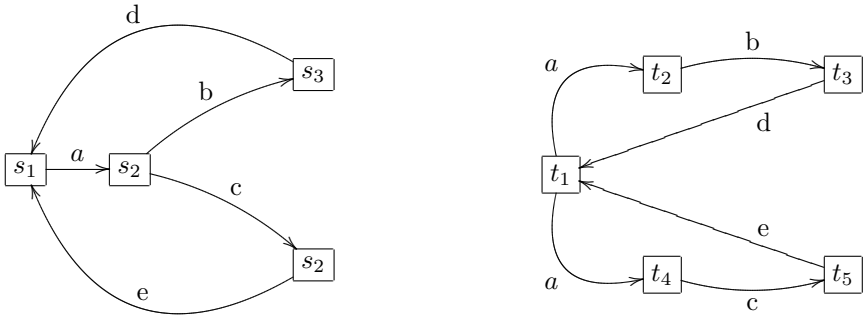


Fig. 0.3. Example for trace equivalence

distinguished state designed as *initial*, and a set of distinguished states designed as *final*. Automata theory is important and well established in Computer Science; it is therefore worth pausing on it for a moment, to see how the question of equality of behaviours is treated there.

Automata are string recognisers. A string, say $a_1 \dots a_n$, is accepted by an automaton if its initial state has a derivative under $a_1, \dots, a_n$ that is among the final states. Two automata are equal if they accept the same language, i.e., the same set of strings. (See, any textbood on automata for details on automata theory.)

The analogous equivalence on processes is called *trace equivalence*. It equates two processes P and Q if they can perform the same finite sequences of transitions; precisely, if P has a sequence $P \xrightarrow{\mu_1} P_1 \dots P_{n-1} \xrightarrow{\mu_n} P_n$ then there should be $Q_1, \dots, Q_n$ with $Q \xrightarrow{\mu_1} Q_1 \dots Q_{n-1} \xrightarrow{\mu_n} Q_n$, and the converse on the transitions from Q. Examples of equivalent automata are given in Figures 0.3 and 0.4, where s_1 and t_1 are the initial states, and for simplicity we assume that all states are final. As processes, s_1 and t_1 are indeed trace equivalent.

These equalities are reasonable and natural on automata.

But processes are used in a quite different way with respect to automata. For instance, a string is considered "accepted" by an automaton if the string

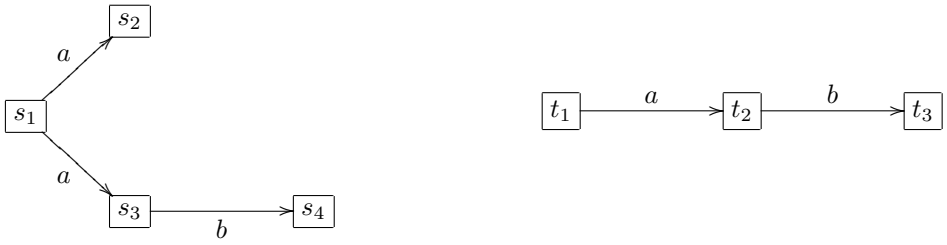


Fig. 0.4. Another example for trace equivalence

gives us *at least one* path from the initial state to a final state; the existence of other paths that fail (i.e., that lead to non-final state) is irrelevant. This is crucial for the equalities in Figures 0.3 and 0.4. For instance, in Figure 0.4, the automaton on the left has a successful path for the string ab, in which the bottom a-transition is taken. But it has also a failing path, along the upper a-transition. In contrast, the automaton on the right only has a successful path. Such differences matter when we interpret the machines as processes. If we wish to press the button a and then the button b of the machine, then our interaction with the machine on the right will always succeed. In contrast, our interaction with the machine on the left may fail. We may indeed reach a deadlock, in which we try to press the button b but the machine refuses such interaction. We cannot possibly consider two processes "the same" when one, and only one of them, can cause a deadlock!

As another example, the equality between the two automaton in Figure 0.3 is rewritten in Figure 0.5 using the labels of the vending machine of Figure 0.1. It is certainly not the same to have in an office the first or the second machine! When we insert a coin in the machine on the right, the resulting state can be either t_2 or t_4. We have no control over this: the machine, non-deterministically, decides. At the end, if we want to have a beverage at all, we must accept whatever the machine offers us. In contrast, the machine on the left always leaves us the choice of our favourite beverage. In concurrency, in contrast with automata theory, the timing of a branch in the transition graph can be important.

0.5 Bisimilarity

In the previous sections we saw that the behavioural equality we seek should:

- imply a tighter correspondence between transitions than language equivalence,
- be based on the informations that the transitions convey, and not on the shape of the LTSs (as in LTS isomorphism).

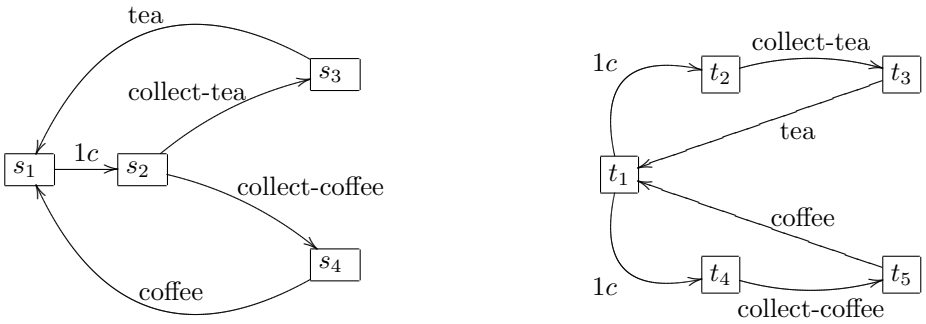


Fig. 0.5. Two vending machines

(Trace equivalence is still important in concurrency. For instance, on deterministic or confluent processes, and for liveness properties such as termination.)

Intuitively, what does it mean that two machines have the same behaviour? If we do something with one machine, then we 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 equality that we are going to formalise. It is called *bisimilarity*.

Definition 0.5.1 (Bisimulation and bisimilarity) A binary relation $\mathcal{R}$ on the states of an LTS is a *bisimulation* if whenever $P_1 \mathcal{R} P_2$:

(1) for all P_1' with $P_1 \xrightarrow{\mu} P_1'$, there is P_2' such that $P_2 \xrightarrow{\mu} P_2'$ and $P_1' \mathcal{R} P_2'$;
(2) the converse, on the transitions emanating from P_2, i.e., for all P_2' with $P_2 \xrightarrow{\mu} P_2'$, there is P_1' such that $P_1 \xrightarrow{\mu} P_1'$ and $P_1' \mathcal{R} P_2'$;

Bisimilarity, written $\sim$, is the union of all bisimulations; thus $P \sim Q$ holds if there is a bisimulation $\mathcal{R}$ with $P \mathcal{R} Q$. □

Note in clause (1) the universal quantifier followed by the existential one: P_1 challenges P_2 on all its transitions, and in each case P_2 is called to find a match.

The definition of bisimilarity immediately suggests a proof technique: to demonstrate that P_1 and P_2 are bisimilar, find a bisimulation relation containing the pair (P_1, P_2). This is the *bisimulation proof method*, and is, by far, the most common method used for proving bisimilarity results. It is useful to get some practice with the method before exploring the theory of bisimilarity.

Remark 0.5.2 Note that bisimulation and bisimilarity are defined on a single LTS, whereas in the previous (informal) examples the processes compared were taken from two distinct LTSs. Having a single LTS is convenient, for instance

ensuring that the alphabet of actions is the same; moreover we do not loose generality, as the union of two LTSs is again an LTS. □

Example 0.5.3 Suppose we want to prove that $s_1 \sim t_1$, for s_i and t_i as in Figure 0.2. We have to find a relation $\mathcal{R}$ containing the pair (s_1, t_1). We thus place (s_1, t_1) in $\mathcal{R}$. For $\mathcal{R}$ to be a bisimulation, we also have to make sure that all (multi-step) derivatives of s_1 and t_1 appear in $\mathcal{R}$—those of s_1 in the first component of the pairs, those of t_1 in the second. Examining s_2, it is natural to place the pair (s_2, t_2) in $\mathcal{R}$. Thus we have $\mathcal{R} = \{(s_1, t_1), (s_2, t_2)\}$. Is this a bisimulation? Obviously not, as a derivative of t_1, namely t_3, is uncovered. Suppose however we did not notice this, and tried to prove that $\mathcal{R}$ is a bisimulation. We have to check clauses (1) and (2) of Definition 0.5.1 on each pair in $\mathcal{R}$. As an example, we consider clause (1) on the pair (s_1, t_1). The only transition from s_1 is $s_1 \stackrel{a}{\longrightarrow} s_2$; this is matched by t_1 via transition $t_1 \stackrel{a}{\longrightarrow} t_2$, for $(s_2, t_2) \in \mathcal{R}$ as required. However the checks on (s_2, t_2) fail, since, for instance, the transition $s_2 \stackrel{b}{\longrightarrow} s_1$ cannot be matched by t_2, whose only transition is $t_2 \stackrel{b}{\longrightarrow} t_3$ and the pair (s_1, t_3) does not appear in $\mathcal{R}$. (Note: if we added to the LTS a transition $t_2 \stackrel{b}{\longrightarrow} t_1$ this problem would disappear, as $(s_1, t_1) \in \mathcal{R}$ and therefore the new transition could now match the challenge from s_2; however $\mathcal{R}$ would still not be a bisimulation; why?) We realise that we have to add the pair (s_1, t_3) to $\mathcal{R}$. We let the reader check that now $\mathcal{R}$ is indeed a bisimulation.

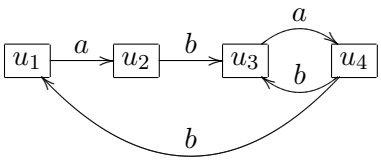
The reader may want to check that the relation $\mathcal{R}$ above remains a bisimulation also when we add the transition $t_2 \stackrel{b}{\longrightarrow} t_1$. □

In the example above, we found a bisimulation after an unsuccessful attempt, which however guided us towards the missing pairs. This way of proceeding is common: trying to prove a bisimilarity $P \sim Q$, one starts with a relation containing at least the pair (P, Q) as an initial guess for a bisimulation; then, checking the bisimulation clauses, one may find that the relation is not a bisimulation because some pairs are missing. The pairs are added, resulting in a new guess for a bisimulation; and so on, until a bisimulation is found.

An important hint to bear in mind when using the bisimulation proof method is to look for bisimulations "as small as possible". A smaller bisimulation, with fewer pairs, reduces the amount of work needed for checking the bisimulation clauses. For instance, in the example above, we could have used, in place of $\mathcal{R}$, the relation $\mathcal{R} \cup \mathcal{I} \cup \{(t_1, t_3)\}$, where $\mathcal{I}$ is the identity relation. This also is a bisimulation, and contains the pair $\{(s_1, t_1)\}$ we are interested in, but it has more pairs and therefore requires more work in proofs. Reducing the size of the relation to exhibit, and hence relieving the proof work needed to establish

bisimilarity results, is the motivation for the enhancements of the bisimulation proof method (so called "up-to techniques", which we will not deal here)

Example 0.5.4 Suppose we want to prove $t_1 \sim u_1$, for t_1 as in Figure 0.2 and u_1 as below.



Proceeding as Example 0.5.3, our initial guess for a bisimulation is the following relation:

$$\{(t_1, u_1), (t_2, u_2), (t_3, u_3), (t_2, u_4)\}$$

This may seem reasonable, as all the states in the LTS are covered. However, this relation is not a bisimulation: clause (2) of Definition 0.5.1 fails on the pair (t_2, u_4), for the transition $u_4 \xrightarrow{b} u_1$ has no match from t_2. We thus add the pair (t_3, u_1). The reader may check that this produces a bisimulation. □

Example 0.5.5 Suppose we want to prove that processes s_1 and t_1 in Figure 0.4 are not bisimilar. We can show that no bisimulations exist that contain such a pair. Suppose $\mathcal{R}$ were such a bisimulation. Then it should also relate the derivative s_2 of s_1 to a derivative of t_1; the only possible such derivative is t_2; but then, on the pair (s_2, t_2), clause (2) of Definition 0.5.1 fails, as only t_2 has a transition.

A similar, but often more useful, method for proving results of non-bisimilarity will be shown in Section 0.11.2, using the approximants of bisimilarity. □

Two features of the definition of bisimulation make its proof method practically interesting:

- the *locality* of the checks on the states;
- the lack of a *hierarchy* on the pairs of the bisimulation.

The checks are local because we only look at the immediate transitions that emanate from the states. An example of a behavioural equality that is non-local is *trace equivalence* (that we encountered when discussing automata). It is non-local because computing a sequence of transitions starting from a state s may require examining other states, different from s.

There is no hierarchy on the pairs of a bisimulation in that no temporal order on the checks is required: all pairs are on a par. As a consequence, bisimilarity can be effectively used to reason about infinite or circular objects. This is in

sharp contrast with inductive techniques, that require a hierarchy, and that therefore are best suited for reasoning about finite objects. For instance, here is a definition of equality that is local but inherently inductive:

> $P_1 = P_2$ if: for all P_1' with $P_1 \xrightarrow{\mu} P_1'$, there is P_2' such that $P_2 \xrightarrow{\mu} P_2'$ and $P_1' = P_2'$; plus the converse, on the transitions from P_2.

This definition requires a hierarchy, as the checks on the pair (P_1, P_2) must follow those on derivative pairs such as (P_1', P_2'). Hence the definition is ill-founded if the state space of the derivatives reachable from (P_1, P_2) is infinite or includes loops. We shall find hierarchical characterisations of $\sim$, refining the idea above, in Section 0.11.2.

Some (very) basic properties of bisimilarity are exposed in Theorems 0.5.6 and 0.5.7.

Theorem 0.5.6 (1) $\sim$ is an equivalence relation, i.e. the following hold:

- (a) $p \sim p$ (reflexivity)
- (b) $p \sim q$ implies $q \sim p$ (symmetry)
- (c) $p \sim q$ and $q \sim r$ imply $p \sim r$ (transitivity);

(2) $\sim$ itself is a strong bisimulation.

Proof For reflexivity, one shows that the identity relation, that is the relation $\{(P, P) \mid P \text{ is a process}\}$, is a bisimulation.

For symmetry, we have to show that if $\mathcal{R}$ is a bisimulation then so is its converse $\mathcal{R}^{-1}$.

For transitivity, we must show that if R_1 and R_2 are bisimulations, then so is their relational composition. *This proof need to be expanded, to show some details* □

The second item of Theorem 0.5.6 brings us the impredicative flavor of the definition of bisimilarity: bisimilarity itself is a bisimulation and is therefore part of the union from which it is defined. The item thus also gives us:

Theorem 0.5.7 $\sim$ is the largest bisimulation, i.e., the largest relation $\sim$ on processes such that $P_1 \sim P_2$ implies:

(1) for all P_1' with $P_1 \xrightarrow{\mu} P_1'$, there is P_2' such that $P_2 \xrightarrow{\mu} P_2'$ and $P_1' \sim P_2'$;
(2) for all P_2' with $P_2 \xrightarrow{\mu} P_2'$, there is P_1' such that $P_1 \xrightarrow{\mu} P_1'$ and $P_1' \sim P_2'$.

0.5.1 Towards coinduction

The assertion in Theorem 0.5.7 could even be taken as the definition of $\sim$ (though we should first show that the largest relation mentioned in the statement does exist). It looks however like a circular definition. Does it make sense? Also, 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?

There is a sharp contrast with the usual, familiar *inductive definitions* and *inductive proofs*. In the case of induction, there is always a basis, i.e., something to start from, and then, in the inductive part, one builds on top of what one has so obtained so far. Indeed, the above definition of $\sim$, and its proof technique, are not inductive, but *coinductive*.

It is good to stop for a while, to get a grasp of the meaning of coinduction, and a feeling of the duality between induction and coinduction. This will be useful to relate the idea of bisimilarity to other concepts, and it will also allow to derive a few results for bisimilarity.

0.6 Examples of induction and coinduction

We begin with some examples, described informally, in which we contrast induction and coinduction.

0.6.1 Finite traces and ω-traces on processes

As an example of an inductive definition, we consider a property akin to termination. A *inactive* process cannot do any transitions. A process P has *a finite trace*, written $P \downarrow$, if P has a finite sequence of transitions that lead to an inactive process as final derivative. Predicate $\downarrow$ has a natural inductive definition, using the following rules:

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

A process P has a finite trace if P is generated by the rules above, in the usual inductive way. An equivalent formulation is to say that $\downarrow$ is the *smallest* set of processes that is *closed forward under the rules*; i.e., the smallest subset S of Pr s.t.

- all inactive processes are in S;
- if there is μ s.t. $P \xrightarrow{\mu} P'$ and $P' \in S$, then also $P \in S$.

This formulation gives us a proof principle for $\lfloor$: given a predicate $\mathcal{P}$ on the processes, to prove that all processes in $\lfloor$ are also in $\mathcal{P}$ it suffices to show that $\mathcal{P}$ is closed forward under the above rules. This is the familiar inductive proof method for sets generated by rules.

As an example of a coinductive definition we consider a property akin to non-termination. Informally, a process P has an *ω-trace under a*, where a is an action (more simply, an ω-trace, when a is clear), if it is possible to observe an infinite sequence of a-transitions starting from P.

Note: process can be both in $\lfloor$ and in $\lceil_a$

The set of processes with an ω-trace has a natural *coinductive* definition in terms of rules. We only need the following inference rule:

$$\frac{P \xrightarrow{a} P' \qquad P' \lceil_a}{P \lceil_a}$$

Indeed $\lceil_a$ is the *largest* predicate on processes that is *closed backward under the rule*; i.e., the largest subset S of processes s.t. if $P \in S$ then

- there is $P' \in S$ s.t. $P \xrightarrow{a} P'$.

Hence: to prove that a given process P has an ω-trace it suffices to find some $S \subseteq Pr$ that is closed backward and with $P \in S$; this is the coinduction proof principle, for ω-traces.

In the first example, the term "closed forward" is to remind us that we are using the rules top-down, from the premises to the conclusion: if S is closed forward, then whenever the premises of a rule are satisfied by S, the resulting conclusion should be in S too. Dually, the term "closed backward" emphasizes that we use the rules bottom-up: if S is closed backward, then each element of S must match a conclusion of a rule in such a way that its premises are satisfied by S.

Of course, the existence of the smallest set closed forward, or the largest set closed backward, must be established. This will follow from the general framework of induction and coinduction that will be introduced later. It is easy however also to prove the existence directly, in each example; for sets closed forward, showing that one such set exists (in the example, the set of all processes), and that the intersection of sets closed forward is again a set closed forward; similarly for sets closed backward.

0.6.2 Reduction to a value and divergence in the λ-calculus

For readers familiar with the λ-calculus, a variant of the previous examples (and probably more enlightening) can be given using the relations of convergence and

the predicate of divergence on the λ-terms. Readers non familiar with the λ-calculus may safely skip the example.

We recall that the set Λ of λ-terms is given by the following grammar (note: this is an inductive definition!)

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

where, in $\lambda x.\, e$, the construct λx is a binder for the free occurrences of x in e. We omit the standard definitions of free and bound variables. The set Λ^0 of *closed λ-terms* is the subset of Λ whose elements have no free variables; $e\{^{e'}\!/\!_x\}$ is the term obtained from e by replacing its free occurrences of x with e'.

Relation $\Downarrow_n \subseteq \Lambda^0 \times \Lambda^0$ (convergence to a value) for the call-by-name λ-calculus, the simplest form of the λ-calculus, is given by the following two rules:

$$\frac{}{\lambda x.\, e \Downarrow_n \lambda x.\, e} \qquad\qquad \frac{e_1 \Downarrow_n \lambda x.\, e_0 \qquad e_0\{^{e_2}\!/\!_x\} \Downarrow_n e'}{e_1(e_2) \Downarrow_n e'}$$

The pairs of terms we are interested in are those generated by these rules; this is an inductive definition. Equivalently, $\Downarrow_n$ is the *smallest* relation on (closed) λ-terms that is *closed forward* under the rules; i.e., the smallest relation $\mathcal{S} \subseteq \Lambda^0 \times \Lambda^0$ s.t.

- $\lambda x.\, e \;\mathcal{S}\; \lambda x.\, e$ for all abstractions,
- if $e_1 \;\mathcal{S}\; \lambda x.\, e_0$ and $e_0\{^{e_2}\!/\!_x\} \;\mathcal{S}\; e'$ then also $e_1(e_2) \;\mathcal{S}\; e'$.

This immediately gives us a proof method for $\Downarrow$ (an example of the induction proof method): given a relation $\mathcal{R}$ on λ-terms, to prove that all pairs in $\Downarrow$ are in $\mathcal{R}$ it suffices to show that $\mathcal{R}$ is closed forward under the above rules. (What is the largest relation closed forward?)

Similarly, the predicate $\Uparrow^n \subseteq \Lambda^0$ (divergence), in call-by-name λ-calculus, is defined coinductively with the following two rules:

$$\frac{e_1 \Uparrow^n}{e_1(e_2) \Uparrow^n} \qquad\qquad \frac{e_1 \Downarrow_n \lambda x.\, e_0 \qquad e_0\{^{e_2}\!/\!_x\} \Uparrow^n}{e_1(e_2) \Uparrow^n}$$

$\Uparrow^n$ is the *largest* predicate on (closed) λ-terms that is *closed backward* under these rules; i.e., the largest subset D of Λ^0 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 that a given term e is divergent it suffices to find $E \subseteq \Lambda^0$ that is closed backward and with $e \in E$ (an example of the coinduction proof method). (What is the smallest predicate closed backward?)

Exercise 0.6.1 Let $e_1 \stackrel{\text{def}}{=} \lambda x.\, xx$, and $e_2 \stackrel{\text{def}}{=} \lambda x.\, xxx$. Show that the terms $e_1(e_1)$, $e_2(e_2)$, $e_1(e_2)$, and $e_2(e_1)$ are all divergent, using the coinduction proof method (i.e., exhibiting suitable sets that are closed backward under the above rules). □

0.6.3 Lists over a set A

Let A be any set. The set of finite lists with elements from A is the set $\mathcal{L}$ inductively generated by the rules below; i.e., the smallest set closed forward under these rules.

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

In contrast, the set of finite and infinite lists is the set coinductively defined by the rules, i.e., the largest set closed backward under the same rules.

0.7 The duality

From the examples above, although informally treated, the pattern of the duality between induction and coinduction begins to emerge:

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

  The destructors show what we can *observe* of the elements. If we think of the elements as black boxes, then the destructors tell us what we can do with them; this is clear in the case of infinite lists, and also in the definition of bisimulation.

When a definition is given 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 is summarised in the table in Figure 0.6. We have not explained yet the duality between congruence and bisimulation; to be done formally, this would require some machinery that we have not introduced here. We can comment however, intuitively, that the dual of *bisimulation* is a *congruence* because a bisimulation is a relation closed under the "destructors", whereas a congruence is a relation closed under the "constructors". Also, we have not explained yet

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

Fig. 0.6. The duality between induction and coinduction

how induction and coinduction are related to least and greatest fixed points. We do this in the next section, where, in fact, we use fixed-point theory to explain the meaning of induction and coinduction. We will thus be able to show the precise sense in which Examples 0.12.7 are about induction or coinduction, and also give a more formal explanation of the duality of the concepts introduced above.

First, we need to introduce the relevant concepts, from fixed-point theory. It is possible to be more general, working with universal algebras or category theory.

0.8 A glimpse of Lattice Theory

In this section we recall a few important results of lattice theory that will then be used to explain induction and coinduction.

0.8.1 Fixed points in complete lattices

Definition 0.8.1 A *partially ordered set* (or *poset*) is a non-empty set equipped with a relation on its elements that is reflexive, transitive, and antisymmetric (antisymmetric meaning that $x \leq y$ and $y \leq x$ implies $x = y$, where $\leq$ is the relation on the set.) □

We usually indicate the partial order relation of a poset by $\leq$. When a set L with relation $\leq$ is a poset, we often simply say that L is a poset. If $x \leq y$ we sometimes say that x is *below* y, and y is *above* x. We also write $y \geq x$ when $x \leq y$ holds.

An *endofunction* on a set L is a function from L onto itself. We sometimes call the elements of a set *points*.

Definition 0.8.2 Let L be a poset.

- An endofunction F on L is *monotone* if $x \leq y$ implies $F(x) \leq F(y)$.

- For a set $S \subseteq L$, a point $y \in L$ is an *upper bound of* S if $x \leq y$ for all points $x \in S$. □

Turning a poset upside-down (that is, reversing the partial order relation) gives us another poset. Thus statements about a poset have a dual, in which each of the relations $\leq$ and $\geq$ is replaced by the other in the statement. For instance, the dual of an upper bound of S is a *lower bound of* S: a point y with $y \leq x$ for all $x \in S$.

Definition 0.8.3 Let L be a poset.

- The *least* element of a subset $S \subseteq L$ is an element $y \in S$ that is a lower bound of S (this element may not exist; if it exists, then it is unique). The least upper bound of S (that is, an upper bound y with $y \leq z$ for all upper bounds z of S) is also called the *join* of S.
- A point x is a *pre-fixed point* of an endofunction F on L if $F(x) \leq x$. If also the converse holds, thus $F(x) = x$, then x is a *fixed point* of F. Further, x is the *least fixed point* of F if x the least element in the set of fixed points of F. □

The dual of the above definitions gives us the *greatest* element of S, the *meet* of S, a *post-fixed point* of F, and the *greatest fixed point* of F. We write $\bigcup S$ (or $\bigcup_{x \in S} S$) for the join of a subset S of a poset, and $\bigcap S$ (or $\bigcap_{x \in S} S$) for its meet. Note that an element y of a subset S of a poset could have the property that no element $x \in S$ exists with $x \leq y$ without y being the least element of S (in the literature such elements are usually called *minimal*). We write $\mathtt{gfp}(F)$ and $\mathtt{lfp}(F)$ for the greatest and least fixed points of F, respectively.

Definition 0.8.4 A *complete lattice* is a poset with all joins (i.e., all the subsets of the given poset have a join). □

The above implies that a complete lattice has also all meets; see Exercise 0.8.6. Further, taking the join and the meet of the empty set, it implies that there are bottom and top elements. We usually indicate these elements by $\bot$ and $\top$, respectively.

Example 0.8.5 If S is a set, then $\wp(S)$ is a complete lattice, ordering the elements of $\wp(S)$ by means of the set inclusion relation $\subseteq$. In this complete lattice, $\emptyset$ (the empty set) and S are the bottom and top elements; join is given by set union, and meet by set intersection. □

The powerset constructions are the kind of complete lattice we mainly use in the chapter. This explains the union and intersection notations adopted for joins and meets.

Exercise 0.8.6 Show that in the definition of complete lattice the existence of all joins implies the existence of all meets. (As usual, the dual is also true, exchanging meets and joins in the definition of complete lattice.) □

Exercise 0.8.7 For the construction in Exercise 0.8.6 it is necessary to assume that *all* subsets have a join. Suppose L is a poset in which all *pairs* of elements have a join (such a poset is called *lattice*—without the adjective "complete"). Show, by means of a counterexample, that this does not imply that all pairs of elements have a meet. □

Remark 0.8.8 A lattice (as defined in Exercise 0.8.7) is complete if and only if every monotone endofunction on the lattice has a fixed-point. Other characterisations of the difference between lattices and complete lattices exist, see books on lattice theory such as [...] for details. □

Theorem 0.8.9 (Fixed-point Theorem) On a complete lattice, a monotone endofunction has a complete lattice of fixed points. In particular the greatest fixed point of the function is the join of all its post-fixed points, and the least fixed point is the meet of all its pre-fixed points. □

Exercise 0.8.10 This exercises invites the reader to carry out a proof of the Fixed-point Theorem.

(1) Let L be the lattice, and F the monotone endofunction, and S the set of fixed-points of L. Consider a subset $X \subseteq S$, and take the set Y of pre-fixed points that are also upper bounds of X:

$$Y \stackrel{\text{def}}{=} \{y \in L \mid F(y) \leq y \text{ and, } \forall x \in X \; x \leq y\}$$

Take now the meet z of Y (which exists because L is a lattice). Show that this is also the join of X in S. (Hint: This is similar to the proof of Exercise 0.8.6.)

(2) Using the previous result, complete the proof of the theorem. □

For our developments in the book, the second part of the theorem, relating greatest and least fixed points to the sets of post-fixed and pre-fixed points, especially interests us. On complete lattices generated by the powerset construction,

the statement becomes: if $F : \wp(X) \to \wp(X)$ is monotone, then

$$\mathtt{lfp}(F) \stackrel{\text{def}}{=} \bigcap \{S \mid F(S) \subseteq S\}$$

$$\mathtt{gfp}(F) \stackrel{\text{def}}{=} \bigcup \{S \mid S \subseteq F(S)\}$$

Exercise 0.8.11 Give a direct proof of the above statement (which is simpler than the full proof of the Fixed-point Theorem). □

Exercise 0.8.12 Another equivalent formulation of the first part of the Fixed-point Theorem can be given in terms of post-fixed points: the monotone endofunction has a complete lattice of post-fixed points. Similarly for the pre-fixed points. Prove these assertions (Hint: this is similar to the proof of the Fixed-point Theorem.) □

0.8.2 Constructive proofs of the existence of least and greatest fixed-points

The proof of the Fixed-point Theorem 0.8.9 we have seen is not constructive (least fixed point and greatest fixed point of a function are obtained from the sets of its pre- and post-fixed points and we are not told how to find these). Theorems 0.8.16 and 0.8.18 give constructive proofs, by means of iterative schemes. Theorem 0.8.16 uses iteration over the natural numbers, but needs additional hypothesis on the function; Theorem 0.8.18 avoids this by iterating over the ordinals. The main advantage of these iteration schemes is that they give us a means of approximating, and possibly even computing, least fixed point and greatest fixed point. The constructions are indeed at the heart of the algorithms used today for computing these fixed-points, including those for checking bisimilarity. The iteration schemes also offer us an alternative way for reasoning about the fixed points; for instance, on greatest fixed point the scheme is useful to prove that a point is not below the greatest fixed point (see for bisimilarity Example 0.11.13).

Definition 0.8.13 An endofunction on a complete lattice is:[1]

- *continuous* if for all sequences $\alpha_0, \alpha_1 \ldots$ of increasing points in the lattice (i.e., $\alpha_i \leq \alpha_{i+1}$, for $i \geq 0$) we have $F(\bigvee_i \alpha_i) = \bigvee_i F(\alpha_i)$.
- *cocontinuous* if for all sequences $\alpha_0, \alpha_1 \ldots$ of decreasing points in the lattice (i.e., $\alpha_i \geq \alpha_{i+1}$, for $i \geq 0$) we have $F(\bigwedge_i \alpha_i) = \bigwedge_i F(\alpha_i)$. □

[1] In some textbooks, cocontinuitiy is called *lower-continuity*, the dual property *upper-continuity*.

In the remainder of this section, we present the details for greatest fixed points and cocontinuity, as they are related to coinduction. The dual statements, using least fixed points and continuity, also hold.

Exercise 0.8.14 If F is cocontinuous, then it is also monotone. (Hint: take $x \geq y$, and the sequence $x, y, y, y, \ldots$.) □

Exercise 0.8.15 Show that a function can be cocontinuous without being continuous, and conversely. □

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 0.8.16 For a cocontinuous endofunction F on a complete lattice we have:

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

□

Dually, if F is continuous:

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

Exercise 0.8.17 Prove Theorem 0.8.16. □

If F is not cocontinuous, and only monotone, we only have $\mathtt{gfp}(F) \subseteq F^{\cap\omega}(\top)$. Therefore if it happens that $F^{\cap\omega}(\top)$ is a fixed point, then we are sure that it is indeed the gfp. With only monotonicity, to reach the greatest fixed point using inductively-defined relations, we need to iterate over the transfinite ordinals.

Theorem 0.8.18 Let F be a monotone endofunction on a complete lattice L, and define $F^\lambda(\top)$, where λ is an ordinal, as follows:

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

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

Proof Using transfinite induction one proves that $\mathtt{gfp}(F) \subseteq F^\lambda(\top)$ for all λ.

As $\beta < \lambda$ implies $F^\beta(\top) < F^\lambda(\top)$ and L is a set, there must be an ordinal α with $F^\alpha(\top) = F^{\alpha+1}(\top)$. As $\mathtt{gfp}(F) \subseteq F^\alpha(\top)$, we can conclude that $F^\alpha(\top)$ is the gfp. □

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 0.8.16 coincides with those used in Theorem 0.8.18, which explains why the notation is the same.

The proof of Theorem 0.8.18 shows that there is an ordinal α of cardinality less than or equal to that of the lattice such that for $\beta \geq \alpha$ the greatest fixed point of F is $F^\beta(\top)$. That is, at α the function reaches its greatest fixed point; on ordinals larger than α, of course, the function remains on such fixed point. In other words, $F^\lambda(\top)$ returns the greatest fixed point of F for all sufficiently large ordinals λ. In case F is cocontinuous, Theorem 0.8.16 assures us that we can take α to be the first ordinal limit, ω (not counting 0 as an ordinal limit).

Exercise 0.8.19 generalises Theorem 0.8.16, for an arbitrary pre-fixed point in place of $\top$. (It is a generalisation because the top element $\top$ of a complete lattice is a pre-fixed point of any function on the lattice, and, as a top element, it is above all post-fixed points of the lattice).

Exercise 0.8.19 Let F be a cocontinuous endofunction on a complete lattice L, and x a pre-fixed point of F, and let $F^n(x)$, $F^{\cap\omega}(x)$ be defined as before Theorem 0.8.16. Show that:

(1) $F_0(x), F_1(x), \ldots$ is a decreasing sequence, and $F^{\cap\omega}(x)$ a fixed point of F;
(2) $F^{\cap\omega}(x)$ is the greatest fixed point of F that is below x;
(3) this fixed point is also the join of all post-fixed points of F that are below x, i.e.,

$$F^{\cap\omega}(x) = \bigcup \{y \mid y \leq x \text{ and } y \leq F(y)\} .$$

□

Exercise 0.8.20 In the same manner as Exercise 0.8.19 is a generalisation of Theorem 0.12.8, state the corresponding generalisation of Theorem 0.12.8, and then prove it. □

Remark 0.8.21 • Theorem 0.8.16 and 0.8.18, and their dual, just mention least and greatest fixed points. It is possible to give similar constructive proofs, using iteration schemes, of the full statement of the Fixed-point Theorem 0.8.9.

- In Theorem 0.8.16 and 0.8.18, and related results, it is the existence of greatest lower bounds of decreasing sequences of points that matters; the existence of arbitrary meets and joins is not needed. Thus the theorems also hold on structures that are weaker than complete lattices. □

0.9 Induction and coinduction, formally

0.9.1 Inductively and coinductively defined sets

For a complete lattice L whose points are sets (as in the complete lattices obtained by the powerset construction) and an endofunction F on L, the sets

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

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

(the join of the post-fixed points, and the meet of the pre-fixed points) are, respectively, the sets *coinductively defined by* F, and *inductively defined by* F. Hence the following rules hold:

$$\text{if } x < F(x) \text{ then } x < F_{\mathtt{coind}} \tag{0.1}$$

$$\text{if } F(x) \leq x \text{ then } F_{\mathtt{ind}} \leq x \tag{0.2}$$

By the Fixed-point Theorem, we know that, when F is monotone, then $F_{\mathtt{coind}}$ is the greatest fixed point (and the greatest post-fixed point) of F, and dually for $F_{\mathtt{ind}}$. More generally, we know that the join of post-fixed points is itself a post-fixed point, and dually so. And Theorems 0.8.16 and 0.8.18 give us ways of approximants of the fixed-points. Rule (0.1) expresses the coinduction proof principle, and rule (0.2) induction proof principle.

To understand the definitions of induction and coinduction given above, in Section 0.10 we revisit the examples in Section 0.6 in the light of such definitions. The examples in Section 0.6 were expressed by means of rules: rules for generating the elements of an inductive set, or for "observing" a coinductive element. So we first show in what sense a set of rules produces monotone operators on complete lattices.

Remark 0.9.1 It is possible to give coinductive and inductive definitions even for functions F that are not monotone. □

0.9.2 Definitions by means of rules

Given a set X, a *ground rule on* X is a pair (S, x) with $S \subseteq X$ and $x \in X$; it intuitively says that from the premises S we can derive the conclusion x. A *set $\mathcal{R}$ of ground rules on X* is a subset of $\wp(X) \times X$; it allows us to obtain elements of X from subsets of X.

Note that what is usually called an inference rule corresponds, in the above terminology, to a set of rules, namely the set of all instances of the inference rule. As an example, consider the inference rule

$$\frac{e_1 \Uparrow^n}{e_1(e_2) \Uparrow^n}$$

on closed λ-terms (Λ^0) that we saw in Section 0.6.2. As a set of ground rules on Λ^0, this becomes the set of all pairs of the form $(\{e\}, e(e'))$, with $e, e' \in \Lambda^0$. Note also that the first component of a ground rule can be empty, which corresponds to the case of an axiom. In the remainder of the section we often omit the adjective "ground".

A set of rules $\mathcal{R}$ on X yields a monotone endofunction $\Phi_{\mathcal{R}}$, called the *functional of $\mathcal{R}$* (or *rule functional*, when $\mathcal{R}$ is clear), 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 0.9.2 Show that $\Phi_{\mathcal{R}}$ above is indeed monotone. □

The relationship between rule functionals and monotone functions is in fact tight, as the following exercise shows.

Exercise 0.9.3 Show that every monotone operators on the complete lattice $\wp(X)$ can be expressed as the functional of some set of rules. □

As the functional of a set of rules is monotone, by the Fixed-point Theorem it has lfp and gfp. These are the sets *inductively* and *coinductively defined by the rules*. We also get, from (0.1) and (0.2), induction and coinduction proof principles.

However, a rule functional need not be continuous or cocontinous.

To do: add a counterexamples here

We can recover continuity and cocontinuity for rule functionals adding some conditions. Here the duality is less obvious, and needs some care.

Definition 0.9.4 A set of rules $\mathcal{R}$ is *finite in the premises*, briefly FP, if for each rule $(S, x) \in \mathcal{R}$ the premise set S is finite. □

Exercise 0.9.5 Show that if the set of rules $\mathcal{R}$ is FP, then $\Phi_{\mathcal{R}}$ is continuous. □

However, surprisingly at first sight, the statement of Exercise 0.9.5 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 $\Phi(S) = \{b\}$ if there is i with $a_i \in S$, otherwise $\Phi(S) = \emptyset$. Consider now the sequence of decreasing sets $S_0, \ldots, S_n, \ldots$, where

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

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

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

Definition 0.9.6 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 0.9.7 If a set of rules $\mathcal{R}$ is FC, then $\Phi_{\mathcal{R}}$ is cocontinuous. □

Exercise 0.9.8 Prove Theorem 0.9.7. □

Corollary 0.9.9 If a set of rules on X is FC, then $\mathtt{gfp}(F) = F^{\cap\omega}(X)$. □

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

Exercise 0.9.10 Let $\mathcal{R}$ be a set of ground rules, and suppose each rule has a non-empty premise. Show that $\mathtt{lfp}(\Phi_{\mathcal{R}}) = \emptyset$. □

0.10 The examples, continued

0.10.1 Finite traces and ω-traces for processes as fixed points

We show how the predicate $\downarrow$ and $\uparrow_a$, from Section 0.6.1, are obtained for suitable sets of ground rules on the set Pr of all processes. In the case of $\downarrow$, the set of rules is:

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

This yield the following functional:

$$\Phi_{\mathcal{R}_\downarrow}(S) \stackrel{\text{def}}{=} \{P \mid P \text{ is inactive, or there are } P', \mu \text{ with } P' \in S \text{ and } P \stackrel{\mu}{\longrightarrow} P'\}$$

A set "closed forward", as in the terminology of Section 0.6.1, is precisely a pre-fixed point of $\Phi_{\mathcal{R}_\downarrow}$. Thus the smallest set closed forward and the proof technique mentioned in Section 0.6.1 become examples of inductively defined set and of induction proof principle.

Exercise 0.10.1 Show that $P \in \mathtt{lfp}(\Phi_{\mathcal{R}_\downarrow})$ if and only if there is $n \geq 0$, processes $P_1, \ldots, P_n$, and actions $\mu_1, \ldots, \mu_n$ s.t. $P \stackrel{\mu_1}{\longrightarrow} P_1 \ldots \stackrel{\mu_n}{\longrightarrow} P_n$ and P_n is inactive. □

In the case of $\restriction_a$, the the set of rules is:

$$\mathcal{R}_{\restriction_a} \stackrel{\text{def}}{=} \{(\{P'\}, P) \mid P \stackrel{a}{\longrightarrow} P'\}\ .$$

This yield the following functional:

$$\Phi_{\mathcal{R}_{\restriction_a}}(S) \stackrel{\text{def}}{=} \{P \mid \text{there is } P' \in S \text{ and } P \stackrel{a}{\longrightarrow} P'\}$$

Now, with references to Section 0.6: a set that is "closed backward" is a post-fixed point of $\Phi_{\mathcal{R}_{\restriction_a}}$, and the largest set closed backward is the greatest fixed point of $\Phi_{\mathcal{R}_{\restriction_a}}$; similarly, the proof technique for ω-traces is derived from the coinduction proof principle.

Exercise 0.10.2 Show that $\mathtt{gfp}(\mathcal{R}_\downarrow) = Pr$, and $\mathtt{lfp}(\mathcal{R}_{\restriction_a}) = \emptyset$. □

Exercise 0.10.3 Show that $P \in \mathtt{gfp}(\mathcal{R}_{\restriction_a})$ if and only if there are processes P_i $(i \geq 0)$ with $P_0 = P$ and s.t., for each i, $P_i \stackrel{a}{\longrightarrow} P_{i+1}$. □

0.10.2 Reduction to a value and divergence in the λ-calculus as fixed-points

Continuing Section 0.6.2, we show how the convergence and divergence in the λ-calculus ($\Downarrow_n$ and $\Uparrow^n$) can be formulated as least fixed point and greatest fixed point of rule functionals. We only show the definition of the functionals, leaving the remaining details to the reader.

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

$$\begin{array}{rl} \Phi_\Downarrow(S) \stackrel{\text{def}}{=} & \{(e, e') \mid \text{for some } e'',\ e = e' = \lambda x.\, e''\ \} \\ & \bigcup\{(e, e') \mid e = e_1(e_2) \text{ and } \exists\ e_0 \text{ s.t. } (e_1, \lambda x.\, e_0) \in S \text{ and } (e_0\{e_2/x\}, e') \in S\}\ . \end{array}$$

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

$$\begin{array}{rl}\Phi_{\Uparrow}(S) \stackrel{\text{def}}{=} & \{e_1(e_2) \mid e_1 \in S,\} \\ & \bigcup\{e_1(e_2) \mid e_1 \Downarrow_n \lambda x.\, e_0 \text{ and } e_0\{^{e_2}\!/\!_x\} \in S\}.\end{array}$$

0.10.3 Lists over a set A as fixed points

We now consider the rules for lists over a set A in Section 0.6.3. We can take X to be the set of all (finite and infinite) strings with elements from the alphabet $A \cup \{\mathtt{nil}, \mathtt{cons}, (,)\}$. The ground rules are $(\emptyset, \mathtt{nil})$ and, for each $s \in X$ and $a \in A$, the rule $(\{s\}, \mathtt{cons}(a, s))$. The corresponding rule functional $F_{A\mathtt{list}}$ is

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

We indicate with $\mathtt{FinLists}_A$ the set of finite lists over A, and with $\mathtt{FinInfLists}_A$ the set of finite and infinite lists over A.

Exercise 0.10.4 Show that $\mathtt{lfp}(F_{A\mathtt{list}}) = \mathtt{FinLists}_A$, and $\mathtt{gfp}(F_{A\mathtt{list}}) = \mathtt{FinInfLists}_A$. □

From (0.2), we infer: suppose $\mathcal{P}$ is a property on $\mathtt{FinLists}_A$; if $F_{A\mathtt{list}}(\mathcal{P}) \subseteq \mathcal{P}$ then $\mathcal{P} \subseteq \mathtt{FinLists}_A$ (hence $\mathcal{P} = \mathtt{FinLists}_A$). Proving $F_{A\mathtt{list}}(\mathcal{P}) \subseteq \mathcal{P}$ requires proving

- $\mathtt{nil} \in \mathcal{P}$;
- $s \in \mathtt{FinLists}_A \cap \mathcal{P}$ implies $\mathtt{cons}(a, s) \in \mathcal{P}$, for all $a \in A$.

This is the familiar induction proof technique for finite lists.

Remark 0.10.5 We used the set X to "bootstrap", via the powerset construction, thus assuming that X is already given. If we were to introduce X formally, then we could do so by means of another coinductive definition; however we could also introduce X in a standard set-theoretic way, by viewing a string over an alphabet B as a special function from the natural numbers to B, namely a function that is either total (which yields an infinite string) or defined only on an initial segment of the naturals (which yields a finite string).

An alternative would be to define lists using the functional $F_{A\mathtt{list}}$ on the universe of all sets (precisely, non-well-founded sets); this would however take us beyond complete lattices. The universe of all sets is not a complete lattice (because of paradoxes such as Russel's), but the constructions that we have seen for least fixed point and greatest fixed point of monotone functions of complete lattices apply. □

An interesting issue is the proof of equality between lists. For finite lists, this does not present a problem: when a set S is inductively defined by means of some rules, we can prove equality between elements of S proceeding in the usual inductive way; that is, reasoning on the depth of the proof with which the elements have been generated from the rules (possibly via transfinite induction, if the rules are not FP). This method does not apply to coinductively defined sets because here the derivation proofs can have infinite paths (i.e., generate a non-well-founded relation on nodes moving upward, from a node toward one of its children). On coinductively defined sets we can prove equalities adapting the idea of bisimulation that we have earlier examined on LTSs. We show this for $\mathtt{FinInfLists}_A$; the same idea applies to any data type coinductively defined via some rules.

The coinductive definition of a set tells us what can be observed of these elements. We can make this explicit in $\mathtt{FinInfLists}_A$ defining an LTS on top of the lists. The domain of the LTS is the set $\mathtt{FinInfLists}_A$, and transitions are given by the following rule:

$$\mathtt{cons}(a, s) \xrightarrow{a} s$$

The rule says that we can observe the head of a list and the result is its tail. Let $\sim_{A\mathtt{list}}$ be the resulting bisimilarity, as by Definition 0.5.1. We have

Lemma 0.10.6 For $s, t \in \mathtt{FinInfLists}_A$, it holds that $s = t$ if and only if $s \sim_{A\mathtt{list}} t$. □

(The above property is often referred to as *(strong) extensionality for* $\mathtt{FinInfLists}_A$, to indicate that the identity relation is the maximal bisimulation on the set.)

Exercise 0.10.7 Let A be a set, and $\mathtt{map} : ((A \to A) \times \mathtt{FinInfLists}_A) \to \mathtt{FinInfLists}_A$ be defined by the following equation:

$$\begin{array}{rcl} \mathtt{map} f \mathtt{nil} & = & \mathtt{nil} \\ \mathtt{map} f (\mathtt{cons}(a, s)) & = & \mathtt{cons}(f(a), \mathtt{map} f s) \end{array}$$

Prove that such a function exists and is unique. (Hint: Let G be the function with the same type as map and that, given a function f and a list, replaces each element a in the list with $f(a)$; show that G satisfies the equations of map, and that for any other function G' that satisfies the same equations, and for any function $f : A \to A$ and list $s \in \mathtt{FinInfLists}_A$, it holds that $G f s \sim_{A\mathtt{list}} G' f s$.) □

Of course it is not necessary to define a LTS from lists. We can directly define a kind of bisimulation on lists, as follows: A relation $\mathcal{R} \subseteq \wp(\mathtt{FinInfLists}_A \times \mathtt{FinInfLists}_A)$ is a *list bisimulation* if whenever $(s, t) \in \mathcal{R}$ then

(1) $s = \texttt{nil}$ implies $t = \texttt{nil}$;
(2) $s = \texttt{cons}(a, s')$ implies there is t' s.t. $t = \texttt{cons}(a, t')$ and $(s', t') \in R$

Then we obtain $\sim_{A\texttt{list}}$ as the union of all list bisimulations.

Similarly to the example of finite lists, one can treat the best known example of inductive definition: the set of natural numbers is characterised as the smallest set contains 0 and that is closed under the successor function. 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.

0.11 Bisimilarity as a fixed-point

0.11.1 The functional of bisimilarity

To see how bisimulation and its proof method fit the coinductive schema, consider the function $F_\sim : \wp(Pr \times Pr) \to \wp(Pr \times Pr)$ defined by:

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

(1) for all P' with $P \xrightarrow{\mu} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$.
(2) for all Q' with $Q \xrightarrow{\mu} Q'$, there is P' such that $P \xrightarrow{\mu} P'$ and $P' \mathcal{R} Q'$.

We call $F_\sim$ the *functional associated to bisimulation*, for we have:

Theorem 0.11.1 (1) $\sim$ is the greatest fixed point of $F_\sim$;
(2) $\sim$ is the largest relation $\mathcal{R}$ such that $\mathcal{R} \subseteq F_\sim(\mathcal{R})$; thus $\mathcal{R} \subseteq \sim$ for all $\mathcal{R}$ with $\mathcal{R} \subseteq F_\sim(\mathcal{R})$. □

Theorem 0.11.1 is a consequence of the Fixed-point Theorem because the functional associated to bisimulation gives us precisely the clauses of a bisimulation, and is monotone on the complete lattice of the relations on $Pr \times Pr$:

Lemma 0.11.2 • $\mathcal{R}$ is a bisimulation iff $\mathcal{R} \subseteq F_\sim(\mathcal{R})$;
• $F_\sim$ is monotone. □

For such functional $F_\sim$, (0.1) asserts that any bisimulation only relates pairs of bisimilar states.

Exercise 0.11.3 Below, $\mathcal{R}$ and $\mathcal{R}_i$ are relations on the processes of a given LTS. Show that:

(1) if $\mathcal{R}$ is an equivalence relation, then also $F_\sim(\mathcal{R})$ is so;

(2) if, for each i in a set I, relation $\mathcal{R}_i$ is an equivalence relation, then also $\bigcap_{i\in I}\mathcal{R}_i$ is so;
(3) use the points (1) and (2) above and Theorem 0.12.8 to conclude that $F_\sim$ is an equivalence relation. □

A process P is *finite* if there are no infinite sequences $P_1,\ldots,P_n\ldots$ and $\mu_1,\ldots,\mu_n\ldots$ with $P \xrightarrow{\mu_1} P_1\ldots P_{n-1} \xrightarrow{\mu_n} P_n\ldots$.

Exercise 0.11.4 What is the least fixed point of $F_\sim$? Conclude that on finite LTSs (i.e., all processes of the LTS are finite) least fixed point and greatest fixed point of $F_\sim$ coincide. □

0.11.2 Approximants of bisimilarity

We can approximate, and even characterise, coinductively defined sets using the iteration schemes of Theorems 0.8.16 and 0.8.18. In this section we examine the operational meaning of these iterations, and related concepts, in the case of bisimilarity.

Definition 0.11.5 (Stratification of bisimilarity, on the naturals) Let Pr be the states of an LTS. We set:

- $\sim_0 \stackrel{\text{def}}{=} Pr\times Pr$
- $P\sim_{n+1} Q$, for $n\geq 0$, if
  (1) for all P' with $P\xrightarrow{\mu}P'$, there is Q' such that $Q\xrightarrow{\mu}Q'$ and $P'\sim_n Q'$;
  (2) the converse, i.e., whenever for all Q' with $Q\xrightarrow{\mu}Q'$, there is P' such that $P\xrightarrow{\mu}P'$ and $P'\sim_n Q'$.
- $\sim_\omega \stackrel{\text{def}}{=} \bigcap_{n\geq 0}\sim_n$. □

Exercise 0.11.6 (1) Show that $\sim_0,\ldots,\sim_n,\ldots$ is a decreasing sequence of relations.
(2) Show for all $0\leq n\leq\omega$, we have $\sim_n = F_\sim^n(Pr)$ and $\sim^\omega = F_\sim^{\cap\omega}(Pr)$, where $F_\sim^n$ and $F_\sim^{\cap\omega}$ are the iterations of $F_\sim$ following the definitions used in Theorem 0.8.16. □

The characterisation in Theorem 0.8.16 required cocontinuity. In general the functional of bisimilarity is not cocontinuous; and $\sim_\omega$ does not coincide with $\sim$, as the following example shows.

Example 0.11.7 Suppose $a \in Act$, and let a^0 be a state with no transitions, a^ω a state whose only transition is

$$a^\omega \xrightarrow{a} a^\omega \ ,$$

and a^n, for $n \geq 1$, states with only transitions

$$a^n \xrightarrow{a} a^{n-1} \ .$$

Also, let P, Q be states with transitions

$$P \xrightarrow{a} a^n \qquad \text{for all } n \geq 0$$

and

$$\begin{array}{ll} Q \xrightarrow{a} a^n & \text{for all } n \geq 0 \\ Q \xrightarrow{a} a^\omega & \end{array}$$

It is easy to prove, by induction on n, that, for all n, $P \sim_n Q$, hence also $P \sim_\omega Q$. However, it holds that $P \not\sim Q$: the transition $Q \xrightarrow{a} a^\omega$ can only be matched by P with one of the transitions $P \xrightarrow{a} a^n$. But, for all n, we have $a^\omega \not\sim a^n$, as only from the former state $n+1$ transitions are possible. □

Exercise 0.11.8 Use Example 0.11.7 to show formally that the function $F_\sim$ is not cocontinuous. □

Remark 0.11.9 A case in which the function F of which bisimilarity is the gfp is usually not cocontinuous is that of *weak* bisimilarity, that we shall introduce later. □

We can obtain $\sim$ by iteration over the natural numbers if we add some conditions to the LTS.

Definition 0.11.10 An LTS is *image-finite* (or *finite-branching*), if, for all P, the set $\{P' \mid P \xrightarrow{\mu} P', \text{ for some } \mu\}$ is finite. □

In Example 0.12.7, the LTS is not image-finite. It becomes image-finite if we remove all transitions $P \xrightarrow{a} a^n$ and $Q \xrightarrow{a} a^n$, for all $n \geq m$, where m is any given number. The LTSs in Figures 0.1-0.5 are image-finite (as the LTS itself has only a finite number of states).

Theorem 0.11.11 On image-finite LTSs, relations $\sim$ and $\sim_\omega$ coincide.

Proof The inclusion $\sim \subseteq \sim_\omega$ is easy: one proves that $\sim \subseteq \sim_n$ for all n, using the fact that $\sim$ is a bisimulation (or, using the fact that $\sim$ is a fixed point of $F_\sim$, monotonicity of $F_\sim$, and the fact that $\sim_{n+1} = F_\sim(\sim_n)$).

Now the converse. We show that the set

$$\mathcal{R} \stackrel{\text{def}}{=} \{(P,Q) \mid P \sim_\omega Q\}$$

is a bisimulation. Thus, take $(P,Q) \in R$, and suppose $P \stackrel{\mu}{\longrightarrow} P'$. We need a matching transition from Q. For all n, as $P \sim_{n+1} Q$, there is Q_n s.t. $Q \stackrel{\mu}{\longrightarrow} Q_n$ and $P \sim_n Q_n$. However, as the LTS is image-finite, the set $\{Q_i \mid Q \stackrel{\mu}{\longrightarrow} Q_i\}$ is finite. Therefore there is at least a Q_i for which $P \sim_n Q_i$ holds for infinitely many n. As the relations $\{\sim_n\}_n$ are decreasing, $P \sim_n Q_i$ holds for all n. Hence $P \sim_\omega Q_i$ and therefore $(P,Q_i) \in \mathcal{R}$. □

Exercise 0.11.12 gives another proof of Theorem 0.11.11, appealing to the cocontinuity of $F_\sim$ and Theorem 0.8.16 (see also Exercise 0.11.17). The proof technique used in the direct proof above is however a useful one to know.

Exercise 0.11.12 Show that under the image-finite hypothesis the functional $F_\sim$ is cocontinuous. □

Example 0.11.13 (Continues Example 0.5.5) The approximants of bisimilarity can be usefully employed to prove non-bisimilarity results. For a (very simple) example, we revisite Example 0.5.5 and show that the processes s_1 and t_1 in Figure 0.4 are not bisimilar. We have to find n such that $s_1 \not\sim_n t_1$. We construct the relations $\sim_i$, starting from 0 and going up. For $i = 0$, we have $Pr \times Pr$. At $i = 1$ we have the pairs of states with the same labels in their immediate transitions: $(s_1,t_1),(s_3,t_2),(s_2,t_3),(s_4,t_3)$. Thus $i = 1$ is not sufficient, for (s_1,t_1) is still present. However, $i = 2$ breaks the pair: the transition $s_1 \stackrel{a}{\longrightarrow} s_2$ cannot be matched by t_1, whose only transition is $t_1 \stackrel{a}{\longrightarrow} t_2$ but s_2 and t_2 are not related in $\sim_1$. □

Theorem 0.11.11 can be strengthened, requiring finiteness on single labels rather than on all transitions. An LTS is *image-finite per label*, if, for all P, and for all $a \in Act$, the set $\{P' \mid P \stackrel{a}{\longrightarrow} P'\}$ is finite. When the set of actions Act can be infinite, image-finiteness per label does not imply image-finiteness.

Exercise 0.11.14 Refine the proof of Theorem 0.11.11 to show the result still holds under the weaker hypothesis of image-finiteness per label. □

In general, as by Theorem 0.8.18, in order to reach $\sim$ we need to replace the ω-iteration that defines $\sim_\omega$ with a transfinite iteration, using the ordinal numbers. Following the definitions of the transfinite iteration in Theorem 0.8.18, at

ordinals successor and at 0 the definition of $\sim_\lambda$ (for λ ordinal) is as in Definition 0.11.5; for ordinals limit we have:

$$\sim_\lambda \stackrel{\text{def}}{=} \bigcap_{\beta<\lambda} \sim_\beta \qquad \text{if } \lambda \text{ is an ordinal limit}$$

and $\sim^\infty \stackrel{\text{def}}{=} \bigcap_\lambda \sim^\lambda$.

Theorem 0.11.15 Relations $\sim$ and $\sim^\infty$ coincide.

Proof Follows from Theorem 0.8.18. □

We have seen that every monotone function on the complete lattice $\wp(X)$ can be expressed as the rule functional of some set of rules on X (Exercise 0.9.3). Thus this also applies to $F_\sim$:

Exercise 0.11.16 Instantiate the statement of Exercise 0.9.3 to the case of $F_\sim$, showing precisely what is the set of rules that has $F_\sim$ as its functional. □

Viewing $F_\sim$ as a rule functional, we can derive the cocontinuity of $F_\sim$ (Exercise 0.11.12) for image-finite LTSs as a special case of the more general theorem relating cocontinuity of rule functionals to the FC property, thus also deriving Theorem 0.11.11 from Corollary 0.9.9 (and, in turn, from Theorem 0.8.16).

Exercise 0.11.17 Use Exercise 0.11.16 to show that image-finiteness of the LTS implies FC (and FP) and therefore derive Theorem 0.11.11. □

Whereas for proving bisimilarity results the bisimulation proof method is simpler to use than the approximants of bisimilarity, for non-bisimilarity results the latter often give more direct proofs.

0.12 Proofs and games for induction and coinduction

0.12.1 Proofs of membership

We examine the duality between sets inductively and coinductively defined from a set of ground rules from the point of view of the proofs of the membership of an element to such sets. We assume that the rules are both FP and FC. While the results also hold without these assumptions, they make some techicalities in the proofs of the main results simpler to understand.

The *trees over* X is the set of all trees in which each node is labelled with an element from the set X and, moreover, the labels of the children of a node are pairwise distinct. If $\mathcal{T}$ is such a tree, then the *root* of $\mathcal{T}$ is the only node without a parent.

Remark 0.12.1 *The definition of trees over X above is informal because we assume that the reader knows what is a tree. For a formal definition, we can take a tree over X to be a set of sequences of elements of X, namely all sequences obtained by picking up a node h in the tree and reading the sequence of labels in the path that goes from the root of the tree to h. Precisely, a tree over X is a set $\mathcal{T}$ of non-empty finite sequences of elements in X such that*

(1) there is only one sequence of length one (corresponding to the root of the tree);
(2) if the sequence $x_1 \dots x_{n+1}$ is in $\mathcal{T}$ then also $x_1 \dots x_n$ is in $\mathcal{T}$.

In this formulation, a sequence $x_1 \dots x_n$ uniquely identifies a node of the tree; and the children of this node are identified by the set of sequences $\bigcup_x \{x_1 \dots x_n x\}$.

In the remainder, in proofs we will sometimes refer to this definition of tree. □

For now, the forms of trees allowed is very general: for instance, a node can have infinitely many children; and there can be infinite paths in the tree that start from the root of the tree and continue moving from a node to one of its children. However, we shall see that the trees that we obtain for proofs of rules under the FC and FP conditions are more constrained. Below we usually omit reference to X, and simply call *tree* a tree over X.

A tree is *non-well-founded* if the relation on the nodes that contains a pair of nodes (h, k) if k is the parent of h is non-well-founded (that is, there are nodes h_i, for $i > 0$, with (h_{i+1}, h_i) in the relation, for each i). The tree is *well-founded* is the relation is well-founded. A non-well-founded tree has infinite paths, whereas in a well-founded tree all paths are finite. Referring to the definition of trees in Remark 0.12.1, a tree $\mathcal{T}$ is non-well-founded if there is infinite sequence $x_1 \dots x_i \dots$ whose prefixes (i.e., all finite sequences $x_1 \dots x_i$, $i > 0$) are all in $\mathcal{T}$.

Now, let $\mathcal{R}$ be a set of ground rules. A tree $\mathcal{T}$ is *a proof tree of $x \in X$ under* $\mathcal{R}$ if x is the label of the root of $\mathcal{T}$ and, for each node h with label y, if S is the set of the labels of all children of h, then (S, y) is a rule in $\mathcal{R}$.

We suppose now that $\mathcal{R}$ is both FP and FC. The FP assumption ensures us that a node only has finitely many children; therefore if a proof tree is well-founded then the tree has a finite heigth, that is, there is a bound on the maximal length of paths in the tree. We recall that $\Phi^n_{\mathcal{R}}$ is the n-th iteration of the functional for the rules $\mathcal{R}$.

Lemma 0.12.2 $x \in \Phi^n_{\mathcal{R}}(\emptyset)$ iff there is a proof tree for x whose heigth is less than or equal to n.

Proof Easy. □

Corollary 0.12.3 $x \in \mathtt{lfp}(\Phi_{\mathcal{R}})$ iff there is a well-founded proof tree for x.

Proof We use the FP hypothesis, which allows us to prove the result from Theorem 0.8.16 and Lemma 0.12.2. □

In other words, the set inductively defined by a the rules $\mathcal{R}$ has precisely all those elements which are obtained from well-founded proofs. We show below that, in contrast, the set coinductively defined by $\mathcal{R}$ has the elements obtained from the well-founded *and* non-well-founded proofs.

A node of a tree is *at level* n ($n \geq 0$) if the path from that node to the root of the tree has precisely $n - 1$ nodes. For instance, the root is at level 0, its children at level 1, the children of the children at level 2, and so forth.

A tree $\mathcal{T}$ is *a proof tree of* $x \in X$ *under* $\mathcal{R}$ *up-to stage* n ($n \geq 0$) if x is the label of the root of $\mathcal{T}$ and, for each node h with label y and at a level $m < n$, if S is the set of the labels of all children of h, then (S, y) is a rule in $\mathcal{R}$ (we check that the tree is "correct" only up to the level n).

Lemma 0.12.4 $x \in \Phi^n_{\mathcal{R}}(X)$ iff there is a proof tree for x up-to stage n.

Proof Easy. □

Corollary 0.12.5 $x \in \mathtt{gfp}(\Phi_{\mathcal{R}})$ iff there is a proof tree for x.

Proof As for Corollary 0.12.3, but in this case we use the FP hypothesis and Lemma 0.12.4. □

0.12.2 A game interpretation of induction and coinduction

We conclude this overview of induction and coinduction with a game-theoretic characterisation of sets inductively and coinductively defined from rules. For this, we re-use some of the ideas in the "proof-tree" presentation of Section 0.12.1.

Consider a set of ground rules $\mathcal{R}$ (on X). A *game in* $\mathcal{R}$ involves two players, which we will indicate as `V` (the verifier) and `R` (the refuter), and an element $x_0 \in X$ with which a play of the game begins. `V` attempts to show that a proof tree for x_0 exists, while `R` attempts to show that there is no such proof. A play begins with `V` choosing a set S_0 such that x_0 can be derived from S_0, that is, $(S_0, x_0) \in R$. Then `R` answers by picking up an element $x_1 \in S_0$, thus challenging `V` to continue with the proof on x_1. Now `V` thus has to find a set S_1 with $(S_1, x_1) \in \mathcal{R}$; then `R` picks $x_2 \in S_1$, and so on. Thus *a play for* $\mathcal{R}$ *and* x_0 is a sequence

$$x_0 S_0 \ldots x_n S_n \ldots$$

which can be finite or infinite. If it is finite, then the play may ends with some x_n (meaning that R made the last move) or with some S_n (V moved last).

In the definition of win of a play we have to distinguish induction from coinduction. We write $\mathcal{G}^{\mathtt{ind}}(\mathcal{R}, x_0)$ for the "inductive" game, and $\mathcal{G}^{\mathtt{coind}}(\mathcal{R}, x_0)$ for the "coinductive" game. In both games, when the play is finite, and one of the player is supposed to make a move but he/she is unable to do so, then the other player wins. This occus if V's last move was the empty set $\emptyset$; V wins because R has no further element to throw in. The end of the game also occurs if R's last move was an element x that does not appear in conclusions of the rules $\mathcal{R}$, in which case R is the winner. The difference between induction and coinduction is in the interpretation of wins for infinite plays. In the inductive world an infinite play is a win for R. This because, as seen in Section 0.12.1, the proof of an element of an inductive set must be well-founded, and infinite plays represent non-well-founded paths in the proof tree. In contrast, in the coinductive world an infinite play is a win for V as here non-well-founded paths in proof trees are allowed.

Example 0.12.6

In any game, however, one of the two players has the possibility of carefully chosing his/her move so to win all plays, irrespectively of the other player's moves. We say that the winning player *has a winning strategy* in the game, that is, a systematic way of playing that will produce a win in every run of the game.

A *strategy for* V in a game $\mathcal{G}^{\mathtt{coind}}(\mathcal{R}, x_0)$ or $\mathcal{G}^{\mathtt{ind}}(\mathcal{R}, x_0)$ is a function that associates to each play

$$x_0 S_0 \ldots x_n S_n x_{n+1}$$

a set S_{n+1} to be used for the next move for V; similarly, a *strategy for* R in $\mathcal{G}^{\mathtt{coind}}(\mathcal{R}, x_0)$ or $\mathcal{G}^{\mathtt{ind}}(\mathcal{R}, x_0)$ is a function that associates to each play

$$x_0 S_0 \ldots x_n S_n$$

an element x_{n+1}. (The strategies we need for inductively and coinductively defined sets can actually be history-free, meaning that the move of a player is dictated only by the last move from the other player, as opposed to the entire play as we have defined above.)

Example 0.12.7 Show the winning strategy in the previous example, with finite traces

Theorem 0.12.8 (1) $x_0 \in \mathtt{lfp}(\Phi_{\mathcal{R}})$ iff player V has winning strategy in the game $\mathcal{G}^{\mathtt{ind}}(\mathcal{R}, x)$;

(2) $x_0 \in \mathtt{gfp}(\Phi_{\mathcal{R}})$ iff player V has winning strategy in the game $\mathcal{G}^{\mathtt{coind}}(\mathcal{R}, x)$.

Proof We examine (1), as (2) is similar. We appeal to Corollaries 0.12.3 and 0.12.5, and the representation of trees as sets of sequences in Remark 0.12.1. Consider all the plays

$$x_0 S_0 \dots x_n S_n$$

that can be obtained following the winning strategy of V. Each such play gives us a sequence

$$x_0 \dots x_n$$

The set of all these sequences is a proof tree for x_0. It is easy to check that all conditions of a tree and a proof tree hold.

The converse is proved similarly, defining a winning strategy from a proof tree for x_0. □

Exercise 0.12.9 Extract the resulting game interpretation of bisimilarity □