Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

May 1, 2016

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Preface

The origins of this book lie their roots on more than 15 years of teaching a course on formal semantics to graduate Computer Science to students in Pisa, originally called *Fondamenti dell'Informatica: Semantica* (*Foundations of Computer Science: Semantics*) and covering models for imperative, functional and concurrent programming. It later evolved to *Tecniche di Specifica e Dimostrazione* (*Techniques for Specifications and Proofs*) and finally to the currently running *Models of Computation*, where additional material on probabilistic models is included.

The objective of this book, as well as of the above courses, is to present different *models of computation* and their basic *programming paradigms*, together with their mathematical descriptions, both *concrete* and *abstract*. Each model is accompanied by some relevant formal techniques for reasoning on it and for proving some properties.

To this aim, we follow a rigorous approach to the definition of the *syntax*, the *typing* discipline and the *semantics* of the paradigms we present, i.e., the way in which well-formed programs are written, ill-typed programs are discarded and the way in which the meaning of well-typed programs is unambiguously defined, respectively. In doing so, we focus on basic proof techniques and do not address more advanced topics in detail, for which classical references to the literature are given instead.

After the introductory material (Part I), where we fix some notation and present some basic concepts such as term signatures, proof systems with axioms and inference rules, Horn clauses, unification and goal-driven derivations, the book is divided in four main parts (Parts II-V), according to the different styles of the models we consider:

IMP:	imperative models, where we apply various incarnations of well-founded induction and introduce λ-notation and concepts like structural recursion, program equivalence, compositionality, completeness and correctness, and also complete partial orders, continuous functions, fixpoint theory;
HOFL:	higher-order functional models, where we study the role of type systems, the main concepts from domain theory and the distinction between lazy and eager evaluation;

CCS, π:	concurrent, non-deterministic and interactive models, where, starting from operational semantics based on labelled transition systems, we introduce the notions of bisimulation equivalences and observational congruences, and overview some approaches to name mobility, and temporal and modal logics system specifications;
PEPA:	probabilistic/stochastic models, where we exploit the theory of Markov chains and of probabilistic reactive and generative systems to address quantitative analysis of, possibly concurrent, systems.

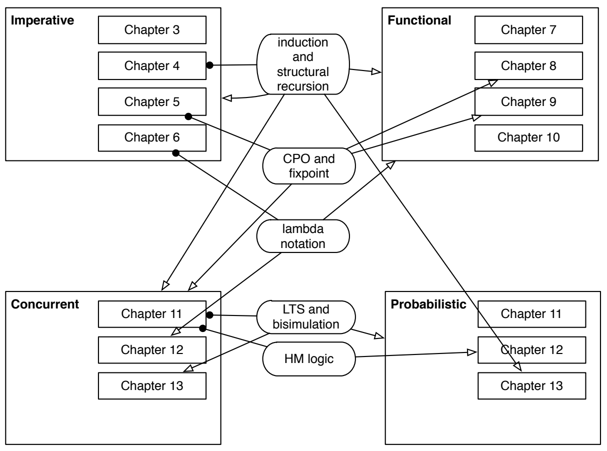
Each of the above models can be studied in separation from the others, but previous parts introduce a body of notions and techniques that are also applied and extended in later parts.

Parts I and II cover the essential, classic topics of a course on formal semantics.

Part III introduces some basic material on process algebraic models and temporal and modal logic for the specification and verification of concurrent and mobile systems. CCS is presented in good detail, while the theory of temporal and modal logic, as well as π-calculus, are just overviewed. The material in Part III can be used in conjunction with other textbooks, e.g., on model checking or π-calculus, in the context of a more advanced course on the formal modelling of distributed systems.

Part IV outlines the modelling of probabilistic and stochastic systems and their quantitative analysis with tools like PEPA. It poses the basis for a more advanced course on quantitative analysis of sequential and interleaving systems.

The diagram that highlights the main dependencies is represented below:



The diagram contains a squared box for each chapter / part and a rounded-corner box for each subject: a line with a filled-circle end joins a subject to the chapter where it is introduced, while a line with an arrow end links a subject to a chapter or part where it is used. In short:

Induction and recursion:	various principles of induction and the concept of structural recursion are introduced in Chapter 4 and used extensively in all subsequent chapters.

CPO and fixpoint:	the notion of complete partial order and fixpoint computation are first presented in Chapter 5. They provide the basis for defining the denotational semantics of IMP and HOFL. In the case of HOFL, a general theory of product and functional domains is also introduced (Chapter 8). The notion of fixpoint is also used to define a particular form of equivalence for concurrent and probabilistic systems, called bisimilarity, and to define the semantics of modal logic formulas.
Lambda-notation:	λ-notation is a useful syntax for managing anonymous functions. It is introduced in Chapter 6 and used extensively in Part III.
LTS and bisimulation:	Labelled transition systems are introduced in Chapter 11 to define the operational semantics of CCS in terms of the interactions performed. They are then extended to deal with name mobility in Chapter 13 and with probabilities in Part V. A bisimulation is a relation over the states of an LTS that is closed under the execution of transitions. The before mentioned bisimilarity is the coarsest bisimulation relation. Various forms of bisimulation are studied in Part IV and V.
HM-logic:	Hennessy-Milner logic is the logic counterpart of bisimilarity: two state are bisimilar if and only if they satisfy the same set of HM-logic formulas. In the context of probabilistic system, the approach is extended to Larsen-Skou logic in Chapter 15.

Each chapter of the book is concluded by a list of exercises that span over the main techniques introduced in that chapter. Solutions to selected exercises are collected at the end of the book.

Pisa, *Roberto Bruni*
February 2016 *Ugo Montanari*

Acknowledgements

We want to thank our friend and colleague Pierpaolo Degano for encouraging us to prepare this book and submit it to the EATCS monograph series. We thank Ronan Nugent and all the people at Springer for their editorial work. We acknowledge all the students of the course on *Models of Computation (MOD)* in Pisa for helping us to refine the presentation of the material in the book and to eliminate many typos and shortcomings from preliminary versions of this text. Last but not least, we thank Lorenzo Galeotti, Andrea Cimino, Lorenzo Muti, Gianmarco Saba, Marco Stronati, former students of the course on *Models of Computation*, who helped us with the LaTeX preparation of preliminary versions of this book, in the form of lecture notes.

Contents

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\approxeq$	CCS weak observational congruence (see Section 11.8.2)
$\cong$	CCS dynamic bisimilarity (see Definition 11.18)
$\overset{\circ}{\sim}_E$	π-calculus early bisimilarity (see Definition 13.3)
$\overset{\circ}{\sim}_L$	π-calculus late bisimilarity (see Definition 13.4)
$\sim_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\sim_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\overset{\bullet}{\approx}_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\overset{\bullet}{\approx}_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\mathrm{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.1.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)

DTMC	Discrete Time Markov Chain (see Definition 14.14)
Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.6)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.6)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.1.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.13)
PI	set of partial injective functions on natural numbers (see Problem 5.12)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.3.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.14)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

Part IV
Concurrent Systems

This part focuses on models and logics for concurrent, interactive systems. Chapter 11 defines the syntax, operational semantics and abstract semantics of CCS, a calculus of communicating systems. Chapter 12 introduces several logics for the specification and verification of concurrent systems, namely LTL, CTL and the μ-calculus. Chapter 13 studies the π-calculus, an enhanced version ofCCS, where new communication channels can be created dynamically and communicated to other processes.

Chapter 11
CCS, the Calculus for Communicating Systems

I think it's only when we move to concurrency that we have enough to claim that we have a theory of computation which is independent of mathematical logic or goes beyond what logicians have studied, what algorithmists have studied. (Robin Milner)

Abstract In the case of sequential paradigms like IMP and HOFL we have seen that all computations are deterministic and that any two non-terminating programs are equivalent. This is not necessary the case for concurrent, interacting systems, which can exhibit different observable behaviours while they compute, also along infinite runs. Consider, e.g., the software governing a web server or the processes of an operating system. In this chapter we introduce a language, called CCS, whose focus is the interaction between concurrently running processes. CCS can be used both as an abstract specification language and as a programming language, allowing seamless comparison between system specifications (desired behaviour) and concrete implementations. We shall see that non-determinism and non-termination are desirable semantics features in this setting. We start by presenting the operational semantics of CCS in terms of a labelled transition system. Then we define some abstract equivalences between CCS terms, and investigate their properties with respect to compositionality and algebraic axiomatisation. We also define a suitable modal logic, called Hennessy-Milner logic, whose induced logical equivalence is shown to coincide with a milestone abstract equivalence, called strong bisimilarity. Finally, we characterise strong bisimilarity as a fixpoint of a monotone operator and explore some alternative abstract equivalences where internal, invisible actions are abstracted away.

11.1 From Sequential to Concurrent Systems

In the last decade computer science technologies have boosted the growth of large scale concurrent and distributed systems. Their formal study introduces several aspects which are not present in the case of sequential programming languages like those studied in previous chapters. In particular, it emerges the necessity to deal with:

Non-determinism:	Non-determinism is needed to model time races between different signals and to abstract away from programming details which are irrelevant for the interaction behaviour of systems.

Parallelism:	Parallelism allows agents to perform tasks independently. For our purposes, this will be modelled by using non-deterministic interleaving of concurrent transitions.
Interaction:	Interaction allows us to describe the behaviour of the system from an abstract point of view (e.g., the behaviour that the system exhibits to an external observer).
Infinite runs:	Accounting for different non-terminating behaviours at the semantic level allows us to distinguish different classes of non-terminating processes, when they have different interaction capabilities.

Accordingly, some additional efforts must be spent to extend in a proper way the semantics of sequential systems to that of concurrent systems.

In this chapter we introduce CCS, a specification language which allows to describe *concurrent communicating systems*. Such systems are composed of *agents* (also *processes*) that communicate through channels.

The semantics of sequential languages can be given by defining functions. In the presence of non-deterministic behaviour functions do not seem to provide the right tool to abstract the behaviour of concurrent systems. As we will see, this problem is worked out by modelling the system behaviour as a *labelled transition system*, i.e., as a set of states equipped with a transition relation which keeps track of the interactions between the system and its environment. Transitions are labelled with symbolic actions that model the kind of computational step that is performed. In addition, recall that the denotational semantics is based on fixpoint theory over CPOs, while it turns out that several interesting properties of non-deterministic systems with non-trivial infinite behaviours are not inclusive (as it is the case of fairness, described in Example 6.9), thus the principle of computational induction does not apply to such properties. As a consequence, defining a satisfactory denotational semantics for CCS is far more complicated than for the sequential case.

Non-terminating sequential programs, as expressed in IMP and HOFL are assigned the same semantics, For example, we recall that, in the denotational semantics, any sequential program that does not terminate is assigned the denotation $\perp$ (e.g., the IMP command **while true do skip** and the HOFL term **rec** $x.\ x$), hence all diverging programs are considered as equivalent. Labelled transition systems allow to assign different semantics to non-terminating concurrent programs.

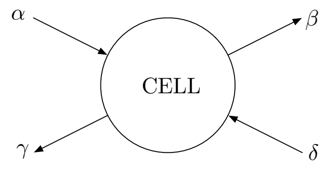
Last, but not least, labelled transition systems are often equipped with a modal logic counterpart, which allows to express and prove the relevant properties of the modelled system.

Let us show how CCS works with an example.

Example 11.1 (Dynamic concurrent stack). Let us consider the problem of modelling an extensible stack. The idea is to represent the stack as a collection of cells that are dynamically created and disposed and that communicate by sending and receiving data over some channels:[1]

- the *send* operation of data v over channel α is denoted by $\overline{\alpha}v$;
- the *receive* operation of data x over channel α is denoted by αx.

We have one process (or agent) for each cell of the stack. Each process can store one incoming value or send a stored value to other processes. All processes involved in the implementation of the extensible stack follow essentially the same communication pattern. We represent graphically one of such processes as follows:



The figure shows that a CELL has four channels $\alpha, \beta, \gamma, \delta$ that can be used to communicate with other cells. A stack is obtained by aligning the necessary cells in a sequence. In general, a process can perform bidirectional operations on its channels. Instead, in this particular case, each cell will use each channel for either input or output operations (but not both) as suggested by the arrows in the above figure:

Channel α:	is the input channel to receive data from either the external environment or the left neighbour cell;
Channel γ:	is the channel used to send data to either the external environment or the left neighbour cell;
Channel β:	is the channel used to send data to the right neighbour cell and to manage the end of the stack;
Channel δ:	is the channel used to receive data from the right neighbour cell and to manage the end of the stack.

In the following, we specify the possible states (CELL_0, CELL_1, CELL_2 and ENDCELL) that a cell can have, each corresponding to some specific behaviour. Note that some states are parametric to certain values that represent, e.g., the particular values stored in that cell. The four possible states are described below.

$$\mathrm{CELL}_0 \stackrel{\text{def}}{=} \delta x.\,\textbf{if}\ x = \$\ \textbf{then}\ \mathrm{ENDCELL}\ \textbf{else}\ \mathrm{CELL}_1(x)$$

The state CELL_0 represents the empty cell. The agent CELL_0 waits for some data from the channel δ and stores it in x. When a value is received the agent checks if it is equal to a special termination character \$. If the received data is \$ this means that the agent is becoming the last cell of the stack, so it switches to the ENDCELL state. Otherwise, if x is a valid value, the agent moves to the state $\mathrm{CELL}_1(x)$.

[1] In the literature, alternative notations for send and receive operations can be found, such as $\alpha!v$ for sending the value v over α and $\alpha?(x)$ or just $\alpha(x)$ for receiving a value over α and binding it to the variable x.

$$\mathrm{CELL}_1(v) \overset{\mathrm{def}}{=} \alpha y.\mathrm{CELL}_2(y,v) \quad + \quad \overline{\gamma} v.\mathrm{CELL}_0$$

The state $\mathrm{CELL}_1(v)$ represents a cell that contains the value v. In this case the cell can non-deterministically wait for new data on α or send the stored data v on γ. In the first case, the cell stores the new value in y and passes the old value v to the agent that models the cell on its right: this task is performed by $\mathrm{CELL}_2(y,v)$. The second case happens when the stored value v is extracted from the cell; then the cell sends the value v on γ and it becomes empty by switching to the state CELL_0. Note that the operator $+$ represents a non-deterministic choice performed by the agent. However a particular choice could be forced on a cell by the behaviour of its neighbours.

$$\mathrm{CELL}_2(u,v) \overset{\mathrm{def}}{=} \overline{\beta} v.\mathrm{CELL}_1(u)$$

The cell in state $\mathrm{CELL}_2(u,v)$ carries two parameters u (the last received value) and v (the previously stored value). The agent must cooperate with its neighbours to shift the data to the right. To this aim, the agent communicates to the right neighbour the old stored value v on β and enters the state $\mathrm{CELL}_1(u)$.

$$\mathrm{ENDCELL} \overset{\mathrm{def}}{=} \alpha z.(\mathrm{CELL}_1(z) \underbrace{\frown \mathrm{ENDCELL}}_{\text{a new bottom cell}}) \quad + \quad \gamma\$.\,\mathbf{nil}$$

The state ENDCELL represents the bottom of the stack. An agent in this state can perform two actions in a non-deterministic way. First, if a new value is received on α (in order to perform a right-bound shift), then the new data is stored in z and the agent moves to state $\mathrm{CELL}_1(z)$. At the same time, a new agent is created, whose initial state is ENDCELL, that becomes the new bottom cell of the stack. Note that we want the newly created agent ENDCELL to be able to communicate with its neighbour $\mathrm{CELL}_1(z)$ only. We will explain later how this can be achieved, when giving the exact definition of the linking operation $\frown$ (see Example 11.3). Informally, the β and δ channels of $\mathrm{CELL}_1(z)$ are linked, respectively, to the α and γ channels of ENDCELL and the communication over them is kept private with respect to the environment: only the channels α and γ of $\mathrm{CELL}_1(z)$ will be used to communicate with neighbours cells and all the other communications are kept local. The second alternative is that the agent can send the special symbol \$ to the left neighbour cell, provided it is able to receive this value. This is possible only if the left neighbour cell is empty (see state CELL_0) and after receiving the symbol \$ on its channel δ it becomes the new ENDCELL. Then the present agent concludes its execution becoming the inactive process **nil**.

Now we will show how the stack works. Let us start from an empty stack. We have only one cell in the state ENDCELL, whose channels β and δ are made private, written $\mathrm{ENDCELL}\backslash\beta\backslash\delta$: no neighbour will be linked to the right side of the cell.

Suppose we want to perform a push operation in order to insert the value 1 in the stack. This can be achieved by sending the value 1 on the channel α to the cell ENDCELL (see Figure 11.1).

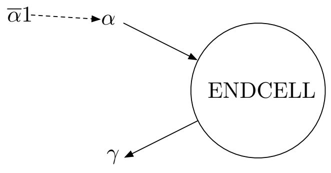


Fig. 11.1: ENDCELL$\backslash\beta\backslash\delta$ receiving the value 1 on channel α

Once the cell receives the new value it generates a new bottom process ENDCELL for the stack and changes its state to $\text{CELL}_1(1)$. The result of this operation is the configuration shown in Figure 11.2.

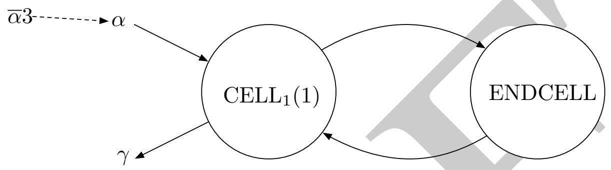


Fig. 11.2: $(\text{CELL}_1(1) \frown \text{ENDCELL})\backslash\beta\backslash\delta$ receiving the value 3 on channel α

When the stack is stabilised we can perform another push operation, say with value 3. In this case the first cell moves to state $\text{CELL}_2(3,1)$ in order to perform a right-bound shift of the previously stored value 1 (see Figure 11.3).

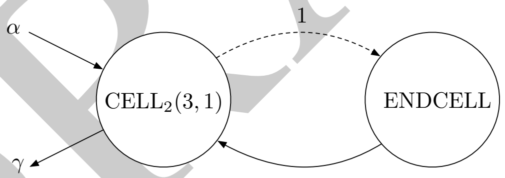


Fig. 11.3: $(\text{CELL}_2(3,1) \frown \text{ENDCELL})\backslash\beta\backslash\delta$ before right-shifting the value 1

Then, when the rightmost cell (ENDCELL) receives the value 1 on its channel α, privately connected to the channel β of the leftmost cell ($\text{CELL}_2(3,1)$) via the linking operation $\frown$, it will change its state to $\text{CELL}_1(1)$ and will spawn a new ENDCELL, while the leftmost cell moves from the state $\text{CELL}_2(3,1)$ to the state $\text{CELL}_1(3)$ (see Figure 11.4).

Now suppose we perform a pop operation, which will return the last value pushed into the stack (i.e., 3). The corresponding operation is an output to the environment (on channel γ) of the leftmost cell. In this case the leftmost cell changes its state

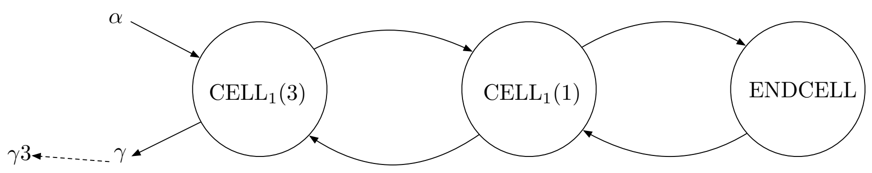


Fig. 11.4: $\text{CELL}_1(3) \frown \text{CELL}_1(1) \frown \text{ENDCELL} \backslash \beta \backslash \delta$ before a pop operation

to CELL_0, and waits for a value through its channel δ (privately connected to the channel γ of the middle cell). The situation is depicted in Figure 11.5.

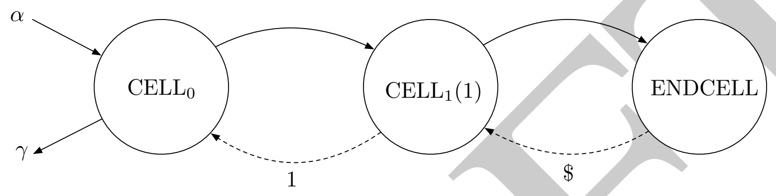


Fig. 11.5: $(\text{CELL}_0 \frown \text{CELL}_1(1) \frown \text{ENDCELL}) \backslash \beta \backslash \delta$ before left-shifting value 1

When the middle cell sends the value 1 to the leftmost cell, it changes its state to CELL_0, and waits for the value sent from the rightmost cell. Then, since the received value from ENDCELL is \$, the middle cell changes its state to ENDCELL, while the rightmost cell reduces to **nil**, as illustrated in Figure 11.6 (where the **nil** agent is just omitted).

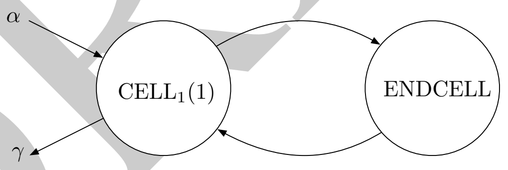


Fig. 11.6: $(\text{CELL}_1(1) \frown \text{ENDCELL} \frown \textbf{nil}) \backslash \beta \backslash \delta$

The above example shows that processes can synchronise in pairs, by performing dual (input/output) operations. In this chapter, we focus on a *pure* version of CCS, where we abstract away from the values communicated on channels. The correspondence with *value passing* CCS is briefly discussed in Section 11.3.8.

11.2 Syntax of CCS

The CCS was introduced by Turing awarded Robin Milner (1934–2010) in the early eighties. We fix the following notation:

$\Delta = \{\alpha, \beta, ...\}$:	denotes the set of channels and, by coercion, input actions;
$\overline{\Delta} = \{\overline{\alpha}, \overline{\beta}, ...\}$:	denotes the set of output actions, with $\overline{\Delta} \cap \Delta = \varnothing$;
$\Lambda = \Delta \cup \overline{\Delta}$:	denotes the set of observable actions;
$\tau \notin \Lambda$:	denotes a distinguished, unobservable action (also called *silent*).

We extend the "bar" operation to all the elements in Λ by letting $\overline{\overline{\alpha}} = \alpha$ for all $\alpha \in \Delta$. As we have seen in the dynamic stack example, pairs of dual actions (e.g., α and $\overline{\alpha}$) are used to synchronise two processes. The unobservable action τ denotes a special action that is internal to some agent and that can no longer be used for synchronisation. Moreover we will use the following conventions:

$\mu \in \Lambda \cup \{\tau\}$:	denotes a generic action;
$\lambda \in \Lambda$:	denotes a generic observable action;
$\overline{\lambda} \in \Lambda$:	denotes the dual action of λ;
$\phi : \Delta \to \Delta$:	denotes a generic permutation of channel names, called a *relabelling*. We extend ϕ to all actions by letting:

$$\phi(\overline{\alpha}) \stackrel{\text{def}}{=} \overline{\phi(\alpha)} \qquad \phi(\tau) \stackrel{\text{def}}{=} \tau.$$

Now we are ready to present the syntax of CCS.

Definition 11.1 (CCS agents). A CCS *agent* (also *process*) is a term generated by the grammar:

$$p, q \quad ::= \quad x \ \mid \ \mathbf{nil} \ \mid \ \mu.p \ \mid \ p\backslash\alpha \ \mid \ p[\phi] \ \mid \ p+q \ \mid \ p \mid q \ \mid \ \mathbf{rec}\ x.\ p$$

We shortly comment the various syntactic elements:

x:	represents a process name;
nil:	is the empty (*inactive*) process;
$\mu.p$:	denotes a process p *prefixed* by the action μ, the process $\mu.p$ can execute μ and become p;
$p\backslash\alpha$:	is a *restricted* process, making the channel α private to p, the process $p\backslash\alpha$ allow synchronisation on α that are internal to p, but disallow external interaction on α;
$p[\phi]$:	is a *relabelled* process that behaves like p after having renamed its channels as indicated by ϕ.
$p+q$:	is a process that can choose non-deterministically to behave as p or q; once the choice is made, the other alternative is discarded;
$p \mid q$:	is the process obtained as the parallel composition of p and q; the actions of p and q can be interleaved and also synchronised;
rec $x.\ p$:	is a recursively defined process, that binds the occurrences of x in p.

As usual, we consider only the closed terms of this language, i.e., all processes such that any process name x always occur under the scope of some recursive definition for x. We name $\mathscr{P}$ the set of closed CCS processes.

11.3 Operational Semantics of CCS

The operational semantics of CCS is defined by a suitable labelled transition system.

Definition 11.2 (Labelled transition system). A *labelled transition system (LTS)* is a triple $(P, L, \rightarrow)$, where P is the set of states of the system, L is the set of labels and $\rightarrow \subseteq P \times L \times P$ is the transition relation. We write $p_1 \xrightarrow{l} p_2$ for $(p_1, l, p_2) \in \rightarrow$.

The LTS that defines the operational semantics of CCS has agents as states and has transitions labelled by actions in $\Lambda \cup \{\tau\}$, denoted by μ. Formally, the LTS is given by $(\mathscr{P}, \Lambda \cup \{\tau\}, \rightarrow)$, where the transition relation $\rightarrow$ is the least one generated by a set of inference rules. The LTS is thus defined by a rule system whose formulas take the form $p_1 \xrightarrow{\mu} p_2$ meaning that the process p_1 can perform the action μ and reduce to p_2. We call $p_1 \xrightarrow{\mu} p_2$ a μ*-transition* of p_1

While the LTS is unique for all CCS processes, when we say "the LTS of a process p" we mean the restriction of the LTS to consider only the states that are reachable from p by a sequence of (oriented) transitions. Although a term can be the parallel composition of many processes, its operational semantics is represented by a single global state in the LTS. Next we introduce the inference rules for CCS.

11.3.1 Inactive Process

There is no rule for the inactive process **nil**: it has no outgoing transition.

11.3.2 Action Prefix

There is only one axiom in the rule system and it is related to action prefix.

$$\text{(Act)} \quad \frac{}{\mu.p \xrightarrow{\mu} p}$$

It states that the process $\mu.p$ can perform the action μ and reduce to p. For example, we have transitions $\alpha.\beta.\,\mathbf{nil} \xrightarrow{\alpha} \beta.\,\mathbf{nil}$ and $\beta.\,\mathbf{nil} \xrightarrow{\beta} \mathbf{nil}$.

11.3.3 Restriction

If the process p is executed under a restriction $\cdot\backslash\alpha$, then it can perform only actions that do not carry the restricted name α as a label.

$$\text{(Res)}\quad \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha}\ \mu \neq \alpha, \overline{\alpha}$$

Note that this restriction does not affect the communication internal to the processes, i.e., when $\mu = \tau$ the move is not blocked by the restriction. For example, the process $(\alpha.\,\mathbf{nil})\backslash\alpha$ is deadlock, while $(\beta.\,\mathbf{nil})\backslash\alpha \xrightarrow{\beta} \mathbf{nil}\backslash\alpha$.

11.3.4 Relabelling

Let ϕ be a permutation of channel names. The μ-transitions of p are renamed to $\phi(\mu)$-transitions by $p[\phi]$.

$$\text{(Rel)}\quad \frac{p \xrightarrow{\mu} q}{p[\phi] \xrightarrow{\phi(\mu)} q[\phi]}$$

We remind that the silent action cannot be renamed by ϕ, i.e., $\phi(\tau) = \tau$ for any ϕ. For example, if $\phi(\alpha) = \beta$, then $(\alpha.\,\mathbf{nil})[\phi] \xrightarrow{\beta} \mathbf{nil}[\phi]$.

11.3.5 Choice

The next pair of rules deals with non-deterministic choice.

$$\text{(Sum)}\quad \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'}$$

Process $p+q$ can choose to behave like either p or q. However, note that the choice can be performed only when an action is performed, e.g., in order to discard the alternative q, the process p must be capable of performing some action μ. For example, if $\phi(\alpha) = \gamma$, $\phi(\beta) = \beta$ and $p \overset{\text{def}}{=} ((\alpha.\,\mathbf{nil} + \overline{\beta}.\,\mathbf{nil})[\phi] + \alpha.\,\mathbf{nil})\backslash\alpha$ we have

$$p \xrightarrow{\gamma} \mathbf{nil}[\phi]\backslash\alpha \qquad \text{and} \qquad p \xrightarrow{\overline{\beta}} \mathbf{nil}[\phi]\backslash\alpha \qquad \text{but not} \qquad p \xrightarrow{\alpha} \mathbf{nil}\backslash\alpha.$$

11.3.6 Parallel Composition

Also in the case of parallel composition some form of non-determinism appears. Unlike the case of sum, where non-determinism is a characteristic of the modelled system, here non-determinism is a characteristic of the semantic style that allows p and q to interleave their actions in $p \mid q$, i.e., non-determinism is exploited to model the parallel behaviour of the system.

$$\text{(Par)} \qquad \frac{p \xrightarrow{\mu} p'}{p \mid q \xrightarrow{\mu} p' \mid q} \qquad \frac{q \xrightarrow{\mu} q'}{p \mid q \xrightarrow{\mu} p \mid q'}$$

The two rules above allows p and q to evolve independently in $p \mid q$. There is also a third rule for parallel composition, which allows processes to perform internal synchronisations.

$$\text{(Com)} \qquad \frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\overline{\lambda}} q_2}{p_1 \mid q_1 \xrightarrow{\tau} p_2 \mid q_2}$$

The processes p_1 and p_2 communicate by using the channel λ in complementary ways. The name of the channel is not shown in the label after the synchronisation by recording the action τ instead.

In general, if p_1 and p_2 can perform α and $\overline{\alpha}$, respectively, then their parallel composition can perform α, $\overline{\alpha}$ or τ. When parallel composition is used in combination with the restriction operator, like in $(p_1 \mid p_2)\backslash\alpha$, then synchronisation on α, if possible, is forced. For example, the LTS for $p \overset{\text{def}}{=} (\alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil}) \mid (\overline{\alpha}.\,\mathbf{nil} + \gamma.\,\mathbf{nil})$ is:

$$\begin{tikzcd}
p \arrow[r, bend left, "\gamma"] \arrow[r, bend right, "\overline{\alpha}"'] \arrow[d, bend right, "\beta"'] \arrow[d, bend left, "\alpha"] \arrow[dr, "\tau"] & (\alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil}) \mid \mathbf{nil} \arrow[d, bend right, "\alpha"'] \arrow[d, bend left, "\beta"] \\
\mathbf{nil} \mid (\overline{\alpha}.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \arrow[r, bend left, "\overline{\alpha}"] \arrow[r, bend right, "\gamma"'] & \mathbf{nil} \mid \mathbf{nil}
\end{tikzcd}$$

while the LTS for process $q \overset{\text{def}}{=} p\backslash\alpha$ is:

$$\begin{tikzcd}
q \arrow[r, bend left, "\gamma"] \arrow[d, bend right, "\beta"'] \arrow[dr, "\tau"] & ((\alpha.\,\mathbf{nil}+\beta.\,\mathbf{nil}) \mid \mathbf{nil})\backslash\alpha \arrow[d, bend left, "\beta"] \\
(\mathbf{nil} \mid (\overline{\alpha}.\,\mathbf{nil}+\gamma.\,\mathbf{nil}))\backslash\alpha \arrow[r, bend right, "\gamma"'] & (\mathbf{nil} \mid \mathbf{nil})\backslash\alpha
\end{tikzcd}$$

When comparing the LTSs for p and q, it is evident that the transitions with labels α and $\overline{\alpha}$ are not present in the LTS for q. Still the τ-labelled transition $q \xrightarrow{\tau} (\mathbf{nil} \mid \mathbf{nil})\backslash\alpha$ that originated from an internal synchronisation over α is present in the LTS of q.

11.3.7 Recursion

The rule for recursively defined processes is similar to the one seen for HOFL terms.

$$\text{(Rec)} \qquad \frac{p[^{\mathbf{rec}\ x.\ p}/_x] \xrightarrow{\mu} q}{\mathbf{rec}\ x.\ p \xrightarrow{\mu} q}$$

The recursive process $\mathbf{rec}\ x.\ p$ can perform all and only the transitions that the process $p[^{\mathbf{rec}\ x.\ p}/_x]$ can perform, where $p[^{\mathbf{rec}\ x.\ p}/_x]$ denotes the process obtained from p by replacing all free occurrences of the process name x with its full recursive definition $\mathbf{rec}\ x.\ p$ (of course, the substitution is capture-avoiding). For example, the possible transitions of the recursive process $\mathbf{rec}\ x.\ \alpha.x$ are the same ones as those of $(\alpha.x)[^{\mathbf{rec}\ x.\ \alpha.x}/_x] = \alpha.\,\mathbf{rec}\ x.\ \alpha.x$, i.e., since

$$\alpha.\,\mathbf{rec}\ x.\ \alpha.x \xrightarrow{\alpha} \mathbf{rec}\ x.\ \alpha.x$$

is the only transition of $\alpha.\,\mathbf{rec}\ x.\ \alpha.x$, there is exactly one transition

$$\mathbf{rec}\ x.\ \alpha.x \xrightarrow{\alpha} \mathbf{rec}\ x.\ \alpha.x.$$

It is interesting to compare the LTSs for the processes below (see Figure 11.7):

$$p \stackrel{\text{def}}{=} (\mathbf{rec}\ x.\ \alpha.x) + (\mathbf{rec}\ x.\ \beta.x) \quad q \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ (\alpha.x + \beta.x) \quad r \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ (\alpha.x + \beta.\,\mathbf{nil})$$

In the first case, p can execute either a sequence of only α-transitions or a sequence of β-transitions. In the second case, q can execute any sequence made of α- and β-transitions. Finally, r admits only sequences of α actions, possibly concluded by a β action. Note that p and q are non-terminating.

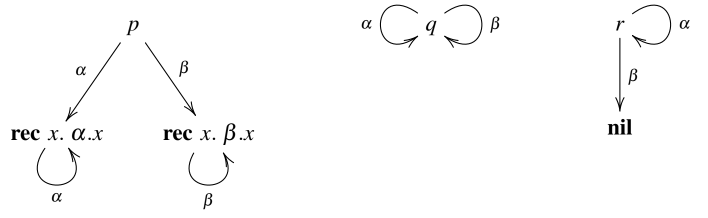


Fig. 11.7: The LTSs of three recursively defined processes

Remark 11.1 (Guarded agents). The form of recursion allowed in CCS is very general. As it is common, we restrict our attention to the class of *guarded agents*, namely agents where, for any recursive sub-terms **rec** $x.\ p$, each free occurrence of x in p occurs under an action prefix (like in all the examples above). This allows us to exclude terms like **rec** $x.\ (x \mid p)$ which can lead (in one step) to an unbounded number of parallel repetitions of the same agent, making the LTS infinitely branching (see Examples 11.12 and 11.13). Formally, given a process p and a set of process names X that must occur guarded in p, we define the predicate $G(p,X)$ as follows:

$$\begin{array}{rlrl} G(\mathbf{nil},X) & \stackrel{\text{def}}{=} \mathbf{true} & G(p\backslash\alpha,X) = G(p[\phi],X) & \stackrel{\text{def}}{=} G(p,X) \\ G(x,X) & \stackrel{\text{def}}{=} x \notin X & G(p+q,X) = G(p \mid q,X) & \stackrel{\text{def}}{=} G(p,X) \wedge G(q,X) \\ G(\mu.p,X) & \stackrel{\text{def}}{=} G(p,\varnothing) & G(\mathbf{rec}\ x.\ p,X) & \stackrel{\text{def}}{=} G(p,X\cup\{x\}) \end{array}$$

The predicate $G(p,X)$ is true if all process names in X and all recursively defined names in p occur guarded in p. A (closed) process p is *guarded* if $G(p,\varnothing)$ holds true. It can be proved that, for any process p and set of process names X:

1. for any process name x: $G(p,X\cup\{x\}) \Rightarrow G(p,X)$, so that, as a particular case, $G(p,X)$ implies $G(p,\varnothing)$; moreover, $G(p,X) \Rightarrow G(p,X\cup\{x\})$ is x does not occur free in p;
2. guardedness is preserved by substitution, namely, for all processes $p_1,...,p_n$ and process names $x_1,...,x_n$:
$$G(p,X) \wedge \bigwedge_{i\in[1,n]} G(p_i,X) \quad \Rightarrow \quad G(p[^{p_1}/_{x_1},\cdots,^{p_n}/_{x_n}],X);$$
3. guardedness is preserved by transitions, namely, for any process q and action μ:
$$G(p,X) \wedge p \xrightarrow{\mu} q \quad \Rightarrow \quad G(q,\varnothing).$$

The proof of items 1 and 2 is by structural induction on p, while the proof of item 3 is by rule induction on $p \xrightarrow{\mu} q$.

Example 11.2 (Derivation). We show an example of the use of the derivation rules we have introduced. Let us take the (guarded) CCS process: $((p \mid q) \mid r)\backslash\alpha$, where:

$$p \overset{\text{def}}{=} \textbf{rec } x.\ (\alpha.x + \beta.x) \qquad q \overset{\text{def}}{=} \textbf{rec } x.\ (\alpha.x + \gamma.x) \qquad r \overset{\text{def}}{=} \textbf{rec } x.\ \overline{\alpha}.x.$$

First, let us focus on the behaviour of the simpler, deterministic agent r. We have:

$$\begin{array}{rl} \textbf{rec } x.\ \overline{\alpha}.x \xrightarrow{\overline{\alpha}} r' & \nwarrow_{\text{Rec}}\ \overline{\alpha}.(\textbf{rec } x.\ \overline{\alpha}.x) \xrightarrow{\overline{\alpha}} r' \\ & \nwarrow_{\text{Act},\ r'=\textbf{rec } x.\ \overline{\alpha}.x}\ \square \end{array}$$

where we have annotated each derivation step with the name of the applied rule. Thus, $r \xrightarrow{\overline{\alpha}} r$ and since there are no other rules applicable during the above derivation, the LTS associated with r consists of a single state and one looping arrow with label $\overline{\alpha}$. Correspondingly, the agent is able to perform the action $\overline{\alpha}$ indefinitely. However, when embedded in the larger system above, then the action $\overline{\alpha}$ is blocked by the topmost restriction $\cdot\backslash\alpha$. Therefore, the only opportunity for r to execute a transition is by synchronising on channel α with either one or the other of the two (non-deterministic) agents p and q. In fact the synchronisation on α produces an action τ which is not blocked by $\cdot\backslash\alpha$. Note that p and q are also available to interact with some external agent on other non-restricted channels (β or γ).

By using the rules of the operational semantics of CCS we have, e.g.:

$$\begin{array}{lrl} ((p \mid q) \mid r)\backslash\alpha \xrightarrow{\mu} s & \nwarrow_{\text{Res},\ s=s'\backslash\alpha} & (p \mid q) \mid r \xrightarrow{\mu} s', \quad \mu \neq \alpha, \overline{\alpha} \\ & \nwarrow_{\text{Com},\ \mu=\tau,\ s'=s'' \mid r_1} & p \mid q \xrightarrow{\lambda} s'', \quad r \xrightarrow{\overline{\lambda}} r_1 \\ & \nwarrow_{\text{Par},\ s''=p \mid q_1} & q \xrightarrow{\lambda} q_1, \quad r \xrightarrow{\overline{\lambda}} r_1 \\ & \nwarrow_{\text{Rec}} & \alpha.q + \gamma.q \xrightarrow{\lambda} q_1, \quad r \xrightarrow{\overline{\lambda}} r_1 \\ & \nwarrow_{\text{Sum}} & \alpha.q \xrightarrow{\lambda} q_1, \quad r \xrightarrow{\overline{\lambda}} r_1 \\ & \nwarrow_{\text{Act},\ \lambda=\alpha,\ q_1=q} & r \xrightarrow{\overline{\alpha}} r_1 \\ & \nwarrow_{\text{Rec}} & \overline{\alpha}.r \xrightarrow{\overline{\alpha}} r_1 \\ & \nwarrow_{\text{Act},\ r_1=r} & \square \end{array}$$

From which we derive:

$$\begin{aligned} r_1 = r &= \textbf{rec } x.\ \overline{\alpha}.x \\ q_1 = q &= \textbf{rec } x.\ \alpha.x + \gamma.x \\ s'' = p \mid q_1 &= (\textbf{rec } x.\ \alpha.x + \beta.x) \mid \textbf{rec } x.\ \alpha.x + \gamma.x \\ s' = s'' \mid r_1 &= ((\textbf{rec } x.\ \alpha.x + \beta.x) \mid (\textbf{rec } x.\ \alpha.x + \gamma.x)) \mid \textbf{rec } x.\ \overline{\alpha}.x \\ s = s'\backslash\alpha &= (((\textbf{rec } x.\ \alpha.x + \beta.x) \mid (\textbf{rec } x.\ \alpha.x + \gamma.x)) \mid \textbf{rec } x.\ \overline{\alpha}.x)\backslash\alpha \\ \mu &= \tau \end{aligned}$$

and thus:
$$((p \mid q) \mid r)\backslash\alpha \xrightarrow{\tau} ((p \mid q) \mid r)\backslash\alpha$$

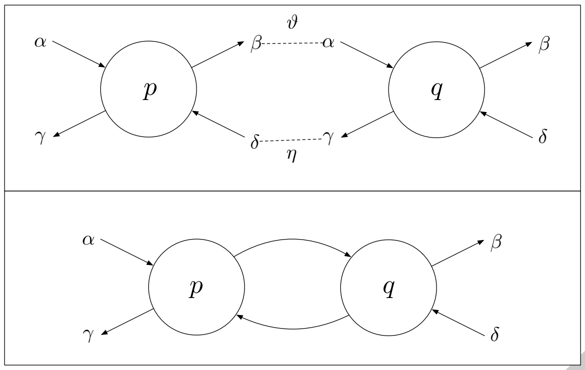


Fig. 11.8: Graphically illustration of the concatenation operator $p \frown q$

Note that during the derivation we had to choose several times between different rules which could have been applied; while in general it may happen that wrong choices can lead to dead ends, our choices have been made so to complete the derivation satisfactorily, avoiding any backtracking. Of course other transitions are possible for the agent $((p \mid q) \mid r)\backslash\alpha$: we leave it as an exercise to identify all of them and draw the complete LTS (see Problem 11.1).

Example 11.3 (Dynamic stack: linking operator). Let us consider again the extensible stack from Example 11.1. We show how to formalise in CCS the linking operator $\frown$. We need two new channels ϑ and η, which will be private to the concatenated cells. Then, we let:

$$p \frown q = (p[\phi_{\beta,\delta}] \mid q[\phi_{\alpha,\gamma}])\backslash\vartheta\backslash\eta$$

where $\phi_{\beta,\delta}$ is the relabelling that sends β to ϑ, δ to η and is the identity otherwise, while $\phi_{\alpha,\gamma}$ sends α to ϑ, γ to η and is the identity otherwise. Notably, ϑ and η are restricted, so that their scope is kept local to p and q, avoiding any conflict on channel names from the outside. For example, messages sent on β by p are redirected to ϑ and must be received by q that views ϑ as α. Instead, messages sent on β by q are not redirected to ϑ and will appear as messages sent on β by the whole process $p \frown q$ (see Figure 11.8).

11.3.8 CCS with Value Passing

Example 11.1 considers i/o operations where values can be received and transmitted. This would correspond to extend the syntax of processes to allow action prefixes like $\alpha(x).p$, where p can use the value x received on channel α and $\overline{\alpha}v.p$, where v is the value sent on channel α. Note that, in this case, x is bound in p. Assuming a set of possible values V as fixed, the corresponding operational semantics rules are:

$$\text{(In)}\quad \frac{v \in V}{\alpha(x).p \xrightarrow{\alpha v} p[^{v}/_{x}]} \qquad \text{(Out)}\quad \frac{}{\overline{\alpha} v.p \xrightarrow{\overline{\alpha} v} p}$$

However, when the set V is finite, we can encode the behaviour of $\alpha(x).p$ and $\overline{\alpha}v.p$ just by introducing as many copies α_v of each channel α as the possible values $v \in V$. If $V = \{v_1, ..., v_n\}$ then:

- an output $\overline{\alpha}v_i.p$ is represented by the process $\overline{\alpha_{v_i}}.p$
- an input $\alpha(x).p$ is represented by the process

$$\alpha_{v_1}.p[^{v_1}/_x] + \alpha_{v_2}.p[^{v_2}/_x] + ... + \alpha_{v_n}.p[^{v_n}/_x].$$

We can also represent quite easily an input followed by a test (for equality) on the received value, like the one used in the encoding of CELL$_0$ in the dynamic stack example: a process like

$$\alpha(x).\,\textbf{if } x = v_i \textbf{ then } p \textbf{ else } q$$

can be represented by the CCS process

$$\alpha_{v_1}.q[^{v_1}/_x] + ... + \alpha_{v_{i-1}}.q[^{v_{i-1}}/_x] + \alpha_{v_i}.p[^{v_i}/_x] + \alpha_{v_{i+1}}.q[^{v_{i+1}}/_x] + ... + \alpha_{v_n}.q[^{v_n}/_x]$$

Example 11.4. Suppose that $V = \{\textbf{true}, \textbf{false}\}$ is the set of booleans. Then a process that waits to receive **true** on the channel α before executing p, can be written as

$$\textbf{rec } x.\ (\alpha_{\textbf{true}}.p + \alpha_{\textbf{false}}.x)$$

11.3.9 Recursive Declarations and the Recursion Operator

In Example 11.1, we have also used recursive declarations, one for each possible state of the cell. They can be expressed in CCS using the recursion operator **rec**. In general, suppose we are given a series of recursive declarations, like:

$$\begin{cases} X_1 \overset{\text{def}}{=} p_1 \\ X_2 \overset{\text{def}}{=} p_2 \\ \quad \cdots \\ X_n \overset{\text{def}}{=} p_n \end{cases}$$

where the symbols $X_1, ..., X_n$ can appear as constants in each of the terms $p_1, ..., p_n$. For any $i \in \{1, ..., n\}$, let

$$q_i \overset{\text{def}}{=} \textbf{rec } X_i.\ p_i$$

be the process where all occurrences of X_i in p_i are bound by the recursive operator (while the instances of X_j occurs freely if $i \neq j$). Then, we can let

$$\begin{aligned}
r_n &\overset{\text{def}}{=} q_n \\
r_{n-1} &\overset{\text{def}}{=} q_{n-1}[^{r_n}/_{X_n}] \\
&\dots \\
r_i &\overset{\text{def}}{=} q_i[^{r_n}/_{X_n}]\dots[^{r_{i+1}}/_{X_{i+1}}] \\
&\dots \\
r_1 &\overset{\text{def}}{=} q_1[^{r_n}/_{X_n}]\dots[^{r_2}/_{X_2}]
\end{aligned}$$

so that in r_i all occurrences of X_j occur under a recursion operator **rec** X_j if $j \geq i$. Then r_1 is a closed CCS process that corresponds to X_1. If we switch the order in which the recursive declarations are listed, the same procedure can be applied to find CCS processes that correspond to the other symbols $X_2, \dots, X_n$.

Example 11.5 (From recursive declarations to recursive processes). For example, suppose we are given the recursive declarations:

$$X_1 \overset{\text{def}}{=} \alpha.X_2 \qquad X_2 \overset{\text{def}}{=} \beta.X_1 + \gamma.X_3 \qquad X_3 \overset{\text{def}}{=} \delta.X_2.$$

Then we have

$$q_1 \overset{\text{def}}{=} \mathbf{rec}\ X_1.\ \alpha.X_2 \qquad q_2 \overset{\text{def}}{=} \mathbf{rec}\ X_2.\ (\beta.X_1 + \gamma.X_3) \qquad q_3 \overset{\text{def}}{=} \mathbf{rec}\ X_3.\ \delta.X_2$$

From which we derive

$$\begin{aligned}
r_3 &\overset{\text{def}}{=} q_3 = \mathbf{rec}\ X_3.\ \delta.X_2 \\
r_2 &\overset{\text{def}}{=} q_2[^{r_3}/_{X_3}] = \mathbf{rec}\ X_2.\ (\beta.X_1 + \gamma.\mathbf{rec}\ X_3.\ \delta.X_2) \\
r_1 &\overset{\text{def}}{=} q_1[^{r_3}/_{X_3}][^{r_2}/_{X_2}] = \mathbf{rec}\ X_1.\ \alpha.\mathbf{rec}\ X_2.\ (\beta.X_1 + \gamma.\mathbf{rec}\ X_3.\ \delta.X_2)
\end{aligned}$$

11.4 Abstract Semantics of CCS

In the previous section we have defined a mapping from CCS agents to LTSs, i.e., to a special class of labelled graphs. It is easy to see that such operational semantics is much more concrete and detailed than the semantics studied for IMP and HOFL. For example, since the states of the LTS are named by agents it is evident that two syntactically different processes like $p \mid q$ and $q \mid p$ are associated with different graphs, even if intuitively one would expect that both exhibit the same behaviour. Analogously for $p+q$ and $q+p$ or for $p+\mathbf{nil}$ and p. Thus it is important to find a good notion of equivalence, able to provide a more abstract semantics for CCS. As it happens for the denotational semantics of IMP and HOFL, an abstract semantics defined *up to equivalence* should abstract away from the syntax and execution details, focusing on some external, visible behaviour. To this aim we can focus on the LTSs associated with agents, disregarding the identity of agents. Another important aspect to be taken into account is compositionality, i.e., the ability to replace any process with an equivalent one inside any context without changing the semantics. Formally, this amounts to define equivalences that are preserved by all the operators of the algebra: they are called *congruences*.

In this section, we first show that neither graph isomorphism nor trace equivalence offer fully satisfactory abstract semantics for CCS. Next, we introduce a more appropriate abstract semantics of CCS by defining a relation, called *strong bisimilarity*, that captures the ability of processes to simulate each other. Finally, we discuss some positive and negative aspects of strong bisimilarity and present some possible alternatives.

11.4.1 Graph Isomorphism

It is quite obvious to require that two agents are equivalent if their (LTSs) graphs are isomorphic. Recall that two labelled graphs are isomorphic if there exists a bijection f between the nodes of the graphs that preserves the graph structure, i.e., such that $v \xrightarrow{\alpha} v'$ iff $f(v) \xrightarrow{\alpha} f(v')$.

Example 11.6 (Isomorphic agents). Let us consider the agents $\alpha.\mathbf{nil} \mid \beta.\mathbf{nil}$ and $\alpha.\beta.\mathbf{nil}+\beta.\alpha.\mathbf{nil}$. Their LTSs are as follows:

$$\begin{tikzcd}
 & \alpha.\mathbf{nil}\mid\beta.\mathbf{nil} \arrow[dl, "\alpha"'] \arrow[dr, "\beta"] \arrow[rrrr, dotted, no head] & & & & \alpha.\beta.\mathbf{nil}+\beta.\alpha.\mathbf{nil} \arrow[dl, "\alpha"'] \arrow[dr, "\beta"] & \\
\mathbf{nil}\mid\beta.\mathbf{nil} \arrow[dr, "\beta"'] \arrow[rrrr, dotted, no head, bend left] & & \alpha.\mathbf{nil}\mid\mathbf{nil} \arrow[dl, "\alpha"] \arrow[rrrr, dotted, no head, bend right] & & \beta.\mathbf{nil} \arrow[dr, "\beta"'] & & \alpha.\mathbf{nil} \arrow[dl, "\alpha"] \\
 & \mathbf{nil}\mid\mathbf{nil} \arrow[rrrr, dotted, no head] & & & & \mathbf{nil} &
\end{tikzcd}$$

The two graphs are isomorphic, as shown by the bijective correspondence represented with dotted lines, thus the two agents should be considered as equivalent. This result is surprising, since they have a rather different structure. In fact, the example shows that concurrency can be reduced to non-determinism by graph isomorphism. This is due to the interleaving of the actions performed by processes that are composed in parallel, which is a peculiar characteristic of the operational semantics which we have presented.

Graph isomorphism is a very simple and natural equivalence relation, but still leads to an abstract semantics that is too concrete, i.e., graph isomorphism distinguishes too much. We show this fact in the following examples.

Example 11.7 (Non-isomorphic agents). Let us consider the (guarded) recursive agents $\mathbf{rec}\ x.\ \alpha.x$, $\mathbf{rec}\ x.\ \alpha.\alpha.x$ and $\alpha.\mathbf{rec}\ x.\ \alpha.x$, whose LTSs are in Figure 11.9:

$$\begin{tikzcd}
\mathbf{rec}\ x.\ \alpha.x \arrow[loop below, "\alpha"] & \mathbf{rec}\ x.\ \alpha.\alpha.x \arrow[d, bend right, "\alpha"'] & \alpha.\mathbf{rec}\ x.\ \alpha.x \arrow[d, "\alpha"'] \\
 & \alpha.\mathbf{rec}\ x.\ \alpha.\alpha.x \arrow[u, bend right, "\alpha"'] & \mathbf{rec}\ x.\ \alpha.x \arrow[loop below, "\alpha"]
\end{tikzcd}$$

Fig. 11.9: Three non-isomorphic agents

The three graphs are not isomorphic, but it is hardly possible to distinguish between the agents according to their behaviour: they all are able to execute any sequence of α-transitions.

Example 11.8 (Buffers). Let us denote by B^n_k a buffer of capacity n of which k positions are busy. For example, for representing a buffer of capacity 1 in CCS one could let (using recursive definitions):

$$B^1_0 \stackrel{\text{def}}{=} in.B^1_1 \qquad B^1_1 \stackrel{\text{def}}{=} \overline{out}.B^1_0$$

The corresponding LTS is

$$\begin{tikzcd}
B^1_0 \arrow[r, bend right, "in"'] & B^1_1 \arrow[l, bend right, "\overline{out}"']
\end{tikzcd}$$

Analogously, for a buffer of capacity 2, one could let:

$$B^2_0 \stackrel{\text{def}}{=} in.B^2_1 \qquad B^2_1 \stackrel{\text{def}}{=} \overline{out}.B^2_0 + in.B^2_2 \qquad B^2_2 \stackrel{\text{def}}{=} \overline{out}.B^2_1$$

Another possibility for obtaining an (empty) buffer of capacity 2 is to use two (empty) buffers of capacity 1 composed in parallel: $B^1_0 \mid B^1_0$. However the LTSs of B^2_0 and $B^1_0 \mid B^1_0$ are not isomorphic, because they have a different number of states:

$$\begin{tikzcd}
B^2_0 \arrow[d, bend right, "in"'] \\
B^2_1 \arrow[u, bend right, "\overline{out}"'] \arrow[d, bend right, "in"'] \\
B^2_2 \arrow[u, bend right, "\overline{out}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
 & B^1_0 \mid B^1_0 \arrow[dl, bend right, "in"'] \arrow[dr, bend left, "in"] & \\
B^1_1 \mid B^1_0 \arrow[ur, "\overline{out}"] \arrow[dr, bend right, "in"'] & & B^1_0 \mid B^1_1 \arrow[ul, "\overline{out}"'] \arrow[dl, bend left, "in"] \\
 & B^1_1 \mid B^1_1 \arrow[ul, "\overline{out}"] \arrow[ur, "\overline{out}"'] &
\end{tikzcd}$$

The LTS of B^2_0 offers a minimal realisation of the behaviour of the buffer: the three states B^2_0, B^2_1 and B^2_2 cannot be identified, because they exhibit different behaviours (e.g., B^2_2 cannot perform an *in* action, unlike B^2_1 and B^2_0, while B^2_0 can perform two *in* actions in a row, unlike B^2_1 and B^2_2). Instead, the LTS of $B^1_0 \mid B^1_0$ has two different states that should be considered as equivalent, namely $B^1_1 \mid B^1_0$ and $B^1_0 \mid B^1_1$ (in our case, it does not matter which position of the buffer is occupied).

11.4.2 Trace Equivalence

A second approach, called *trace equivalence*, observes the set of traces of an agent, namely the set of sequences of actions labelling all paths in its LTS. Trace equivalence is analogous to language equivalence for ordinary automata, except for the fact that in CCS there are no accepting states.

Formally, A *finite trace* of a process p is a sequence of actions $\mu_1 \cdots \mu_k$ (for $k \geq 0$) such that there exists a sequence of transitions

$$p = p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_{k-1}} p_{k-1} \xrightarrow{\mu_k} p_k$$

for some processes $p_1, \ldots, p_k$. Two agents are (finite) trace equivalent if they have the same set of possible (finite) traces. Note that the set of traces associated with one process p is *prefix-closed*, in the sense that if the trace $\mu_1 \cdots \mu_k$ belongs to the set of traces of p, then any of its prefixes $\mu_1 \cdots \mu_i$ with $i \leq k$ also belongs to the set of traces of p.[2] For example, the empty trace ε belongs to the semantics of any process.

Trace equivalence is strictly coarser than equivalence based on graph isomorphism, since isomorphic graphs have the same traces. Conversely, Examples 11.7 and 11.8 show agents which are trace equivalent but whose graphs are not isomorphic. The following example shows that trace equivalence is too coarse: it is not able to capture the choice points within agent behaviour. In the example we exploit the notion of a context.

Definition 11.3 (Context). A *context* is a term with a hole which can be filled by inserting any other term of our language.

We write $C[\cdot]$ to indicate a *context* and $C[p]$ to indicate the context $C[\cdot]$ whose hole is filled with p.

Example 11.9. Let us consider the following agents:

$$p \overset{\text{def}}{=} \alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \qquad q \overset{\text{def}}{=} \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}$$

Their LTSs are as follows:

$$\begin{tikzcd}
\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \arrow[d, "\alpha"] \\
\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil} \arrow[d, bend right, "\beta"'] \arrow[d, bend left, "\gamma"] \\
\mathbf{nil}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil} \arrow[dl, "\alpha"'] \arrow[dr, "\alpha"] & \\
\beta.\,\mathbf{nil} \arrow[dr, "\beta"'] & & \gamma.\,\mathbf{nil} \arrow[dl, "\gamma"] \\
 & \mathbf{nil} &
\end{tikzcd}$$

[2] A variant of trace equivalence, called completed trace semantics, is not prefix-closed and will be discussed in Example 11.15.

The agents p and q are trace equivalent: their set of traces is $\{\varepsilon, \alpha, \alpha\beta, \alpha\gamma\}$. However the agents make their choices at different points in time. In the second agent q the choice between β and γ is made when the first transition is executed, by selecting one of the two outbound α-transitions. In the first agent p, on the contrary, the choice is made on a second time, after the execution of the unique α-transition.

The difference is evident if we consider, e.g., that an agent

$$r \stackrel{\text{def}}{=} \overline{\alpha}.\overline{\beta}.\overline{\delta}.\,\mathbf{nil}$$

is running in parallel, with p or with q, with actions α, β and γ restricted on top: compare

$$(p \mid r)\backslash\alpha\backslash\beta\backslash\gamma \qquad \text{with} \qquad (q \mid r)\backslash\alpha\backslash\beta\backslash\gamma.$$

The agent p is always able to carry out the complete interaction with r, because after the synchronisation on α it is ready to synchronise on β; vice versa, the agent q is only able to carry out the complete interaction with r if the left choice is performed at the time of the first interaction on α, as otherwise $\gamma.\,\mathbf{nil}$ and $\overline{\beta}.\overline{\delta}.\,\mathbf{nil}$ cannot interact. Formally, if we consider the context

$$C[\cdot] = (\cdot \mid \overline{\alpha}.\overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

we have that $C[p]$ and $C[q]$ are trace equivalent, but $C[q]$ can deadlock before executing $\overline{\delta}$, while this is not the case for $C[p]$. Figure out how embarrassing could be the difference if α would mean for a computer to ask the user if a file should be deleted, and β, γ were the user's yes/no answer: p would behave as expected, while q could decide to delete the file in the first place, and then deadlock if the the user decides otherwise. As another example, assume that p and q are possible alternatives for the control of a vending machine, where α models the insertion of a coin and β and γ model the supply of a cup of coffee or a cup of tea: p would let the user choose between coffee and tea, while q would choose for the user. We will consider again processes p and q in Example 11.15, when discussing compositionality issues.

Given all the above, we can argue that neither graph isomorphism nor trace equivalence are good candidates for our behavioural equivalence relation. Still, it is obvious that: 1) isomorphic agents must be retained as equivalent; 2) equivalent agents must be trace equivalent. Thus, our candidate equivalence relation must be situated in between graph isomorphism and trace equivalence.

11.4.3 Strong Bisimilarity

In this section we introduce a class of relations between agents called *strong bisimulations* and we define a behavioural equivalence relation between agents, called *strong bisimilarity*, as the largest strong bisimulation. This equivalence relation is shown to identify only those agents which intuitively have the same behaviour.

Let us start with an example that illustrates how bisimulation works.

Example 11.10 (Bisimulation game). In this example we use game theory in order to show that the agents of the Example 11.9 should not be considered as behaviourally equivalent. Imagine that two opposite players are arguing about the fact that a system satisfies (or not) a given property. One of them, the *attacker*, argues that the system does not satisfy the property. The other player, the *defender*, believes that the system satisfies the property. If the attacker has a winning strategy this means that the system does not satisfy the property. Otherwise, the defender wins, meaning that the system satisfies the property.

The game is turn-based and, at any turn, we let the attacker move first and the defender play back. In the case of bisimulation, the system is composed by two processes p and q and the attacker wants to prove that they are not equivalent, while the defender wants to convince the opponent that p and q are equivalent. Let Alice be the attacker and Bob the defender. The rules of the game are very simple.

Alice starts the game. At each turn:

- Alice chooses one of the processes and executes one of its outgoing transitions.
- Bob must then execute an outgoing transition of the other process, matching the action label of the transition chosen by Alice.
- At the next turn, if any, the game will start again from the target processes of the two transitions selected by Alice and Bob.

If Alice cannot find a move, then Bob wins, since this means that p and q are both deadlock, and thus obviously equivalent. Alice wins if she can make a move that Bob cannot imitate; or if she has a move that, no matter which is the answer by Bob, will lead to a situation where she can make a move that Bob cannot imitate; and so on for any number of moves. Bob wins if Alice has no such a (finite) strategy. Note that the game does not necessarily terminate: also in this case Bob wins, because Alice cannot disprove that p and q are equivalent.

From example 11.9, let us take

$$p \stackrel{\text{def}}{=} \alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \qquad q \stackrel{\text{def}}{=} \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\,.$$

We show that Alice has a winning strategy. Alice starts by choosing p and by executing its unique α-transition $p \xrightarrow{\alpha} \beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}$. Then, Bob can choose one of the two α-transitions leaving from q. Suppose that Bob chooses the α-transition $q \xrightarrow{\alpha} \beta.\,\mathbf{nil}$ (but the case where Bob chooses the other transition leads to the same result of the game). So the processes for the next turn of the game are $\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}$ and $\beta.\,\mathbf{nil}$. At the second turn, Alice chooses the process $\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}$ and the transition $\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil} \xrightarrow{\gamma} \mathbf{nil}$, and Bob can not simulate this move from $\beta.\,\mathbf{nil}$. Since Alice has a winning, two-moves strategy, the two agents are not equivalent.

Now we define the same relation in a more formal way, as originally introduced by Robin Milner. It is important to notice that the definition is not specific to CCS; it applies to a generic LTS $(P, L, \to)$. The labelled transition systems whose states are CCS agents is just a special instance. Below, for $R \subseteq \mathscr{P} \times \mathscr{P}$ a binary relation on agents, we use the infix notation $s_1 \; R \; s_2$ to mean $(s_1, s_2) \in R$.

Definition 11.4 (Strong Bisimulation). Let R be a binary relation on the set of states of an LTS; then it is a *strong bisimulation* if

$$\forall s_1, s_2.\ s_1\ R\ s_2 \Rightarrow \begin{cases} \forall \mu, s_1'.\ \text{ if } s_1 \xrightarrow{\mu} s_1' \text{ then } \exists s_2' \text{ such that } s_2 \xrightarrow{\mu} s_2' \text{ and } s_1'\ R\ s_2' \\ \forall \mu, s_2'.\ \text{ if } s_2 \xrightarrow{\mu} s_2' \text{ then } \exists s_1' \text{ such that } s_1 \xrightarrow{\mu} s_1' \text{ and } s_1'\ R\ s_2'. \end{cases}$$

Trivially, the empty relation is a strong bisimulation and it is easy to check that the identity relation

$$Id \stackrel{\text{def}}{=} \{(p,p) \mid p \in \mathscr{P}\}$$

is a strong bisimulation. Interestingly, graph isomorphism defines a strong bisimulation and the union $R_1 \cup R_2$ of two strong bisimulation relations R_1 and R_2 is also a strong bisimulation relation. The inverse $R^{-1} = \{(s_2, s_1) \mid (s_1, s_2) \in R\}$ of a strong bisimulation R is also a strong bisimulation. Moreover, given the composition of relations defined by

$$R_1 \circ R_2 \stackrel{\text{def}}{=} \{(p,q) \mid \exists r.\ p\ R_1\ r \wedge r\ R_2\ q\}$$

it can be shown that the relation $R_1 \circ R_2$ is a strong bisimulation whenever R_1 and R_2 are such (see Problem 11.4).

Definition 11.5 (Strong bisimilarity $\simeq$). Let s_1 and s_2 be two states of an LTS, then they are said to be *strong bisimilar*, written $s_1 \simeq s_2$ if and only if there exists a strong bisimulation R such that $s_1\ R\ s_2$.

The relation $\simeq$ is called *strong bisimilarity* and is defined as follows:

$$\simeq \quad \stackrel{\text{def}}{=} \quad \bigcup_{R \text{ is a strong bisimulation}} R$$

Remark 11.2. In the literature, strong bisimilarity is often denoted by $\sim$. We use the symbol $\simeq$ to make explicit that it is a congruence relation (see Section 11.5).

To prove that two processes p and q are strong bisimilar it is enough to define a strong bisimulation that contains the pair (p,q).

Example 11.11. Examples 11.7 and 11.8 show agents which are trace equivalent but whose graphs are not isomorphic. Here we show that they are also strong bisimilar. In the case of the agents in Examples 11.7, let us consider the relations

$$R_1 \stackrel{\text{def}}{=} \{(\mathbf{rec}\ x.\ \alpha.x, \mathbf{rec}\ x.\ \alpha.\alpha.x), (\mathbf{rec}\ x.\ \alpha.x, \alpha.\mathbf{rec}\ x.\ \alpha.\alpha.x)\}$$
$$R_2 \stackrel{\text{def}}{=} \{(\mathbf{rec}\ x.\ \alpha.x, \alpha.\mathbf{rec}\ x.\ \alpha.x), (\mathbf{rec}\ x.\ \alpha.x, \mathbf{rec}\ x.\ \alpha.x)\}.$$

In the case of the agents in Example 11.8, let us consider the relation

$$R \stackrel{\text{def}}{=} \{(B_0^2, B_0^1 \mid B_0^1), (B_1^2, B_1^1 \mid B_0^1), (B_1^2, B_0^1 \mid B_1^1), (B_2^2, B_1^1 \mid B_1^1)\}.$$

We invite the reader to check that they are indeed strong bisimulations.

Theorem 11.1 proves that strong bisimilarity $\simeq$ is an equivalence relation on CCS processes. Below we recall the definition of equivalence relation.

Definition 11.6 (Equivalence Relation). Let $\equiv$ be a binary relation on a set X, then we say that it is an *equivalence relation* if it has the following properties:

reflexivity:	$\forall x, y \in X.\ x \equiv x;$
symmetry:	$\forall x, y \in X.\ x \equiv y \Rightarrow y \equiv x.$
transitivity:	$\forall x, y, z \in X.\ x \equiv y \wedge y \equiv z \Rightarrow x \equiv z;$

The equivalence induced by a relation R is the least equivalence that contains R: it is denoted by $\equiv_R$ and is defined by the inference rules below

$$\frac{x\,R\,y}{x \equiv_R y} \qquad \frac{}{x \equiv_R x} \qquad \frac{x \equiv_R y}{y \equiv_R x} \qquad \frac{x \equiv_R y \quad y \equiv_R z}{x \equiv_R z}$$

Note that, in general, a strong bisimulation R is not necessarily reflexive, symmetric or transitive (see, e.g., Example 11.11). However, given any strong bisimulation R, its induced equivalence relation $\equiv_R$ is also a strong bisimulation.

Theorem 11.1. *Strong bisimilarity $\simeq$ is an equivalence relation.*

We omit the proof of Theorem 11.1: it is based on the above mentioned properties of strong bisimulations (see Problem 11.5).

Theorem 11.2. *Strong bisimilarity $\simeq$ is the largest strong bisimulation.*

Proof. We need just to prove that $\simeq$ is a strong bisimulation: by definition it contains any other strong bisimulation. By Theorem 11.1, we know that $\simeq$ is symmetric, so it is sufficient to prove that if $s_1 \simeq s_2$ and $s_1 \xrightarrow{\mu} s_1'$ then we can find s_2' such that $s_2 \xrightarrow{\mu} s_2'$ and $s_1' \simeq s_2'$. Let $s_1 \simeq s_2$ and $s_1 \xrightarrow{\mu} s_1'$. Since $s_1 \simeq s_2$, by definition of $\simeq$, there exists a strong bisimulation R such that $s_1\,R\,s_2$. Therefore, there is s_2' such that $s_2 \xrightarrow{\mu} s_2'$ and $s_1'\,R\,s_2'$. Since $R \subseteq \simeq$ we have $s_1' \simeq s_2'$. □

We can then give a precise characterisation of strong bisimilarity.

Theorem 11.3. *For any states s_1 and s_2 we have:*

$$s_1 \simeq s_2 \quad \Leftrightarrow \quad \begin{cases} \forall \mu, s_1'.\ \textit{if}\ s_1 \xrightarrow{\mu} s_1'\ \textit{then}\ \exists s_2'\ \textit{such that}\ s_2 \xrightarrow{\mu} s_2'\ \textit{and}\ s_1' \simeq s_2' \\ \forall \mu, s_2'.\ \textit{if}\ s_2 \xrightarrow{\mu} s_2'\ \textit{then}\ \exists s_1'\ \textit{such that}\ s_1 \xrightarrow{\mu} s_1'\ \textit{and}\ s_1' \simeq s_2'. \end{cases}$$

Proof. One implication ($\Rightarrow$) follows directly from Theorem 11.2.

The other implication ($\Leftarrow$) is sketched here. Take s_1 and s_2 such that

$$\begin{array}{l} \forall \mu, s_1'.\ \text{if}\ s_1 \xrightarrow{\mu} s_1'\ \text{then}\ \exists s_2'\ \text{such that}\ s_2 \xrightarrow{\mu} s_2'\ \text{and}\ s_1' \simeq s_2' \\ \forall \mu, s_2'.\ \text{if}\ s_2 \xrightarrow{\mu} s_2'\ \text{then}\ \exists s_1'\ \text{such that}\ s_1 \xrightarrow{\mu} s_1'\ \text{and}\ s_1' \simeq s_2'. \end{array}$$

We want to show that $s_1 \simeq s_2$. This is readily done by showing that the relation

$$R \stackrel{\text{def}}{=} \{(s_1,s_2)\} \cup \simeq$$

is a strong bisimulation. By Theorem 11.2, all pairs in $\simeq$ satisfy the requirement fro strong bisimulation. We leave to the reader the task to check that also the pair $(s_1,s_2) \in R$ satisfies the condition. □

Checking that a relation is a strong bisimulation requires checking that all the pairs in it satisfy the condition in Definition 11.4. So it is very convenient to exhibit relations that are as small as possible, e.g., we can avoid to add reflexive, symmetric and transitive pairs, unless needed.

In the following, when we will consider relations that are equivalences, instead of listing all pairs of processes in the relation, we will list just the induced equivalence classes for brevity, i.e., we will work with quotient sets.

Definition 11.7 (Equivalence classes and quotient sets). Given an equivalence relation $\equiv$ on X and an element $x \in X$ we call the *equivalence class* of x the subset $[x]_\equiv \subseteq X$ defined as follows:

$$[x]_\equiv \stackrel{\text{def}}{=} \{y \in X \mid x \equiv y\}$$

The set $X_{/\equiv}$ containing all the equivalence classes generated by a relation $\equiv$ on the set X is called *quotient set*.

11.4.3.1 Strong Bisimilarity as a Fixpoint

Now we re-use fixpoint theory, which we have introduced in the previous chapters, in order to define strong bisimilarity in a more effective way. Using fixpoint theory we will construct, by successive approximations, the coarsest (largest, i.e.. that distinguishes as least as possible) strong bisimulation between the states of an LTS.

As usual, we define the $CPO_\perp$ on which the approximation function works. The $CPO_\perp$ is defined on the set $\wp(\mathscr{P} \times \mathscr{P})$, namely the powerset of the pairs of CCS processes. We know that, for any set S, the structure $(\wp(S), \subseteq)$ is a $CPO_\perp$, but it is not exactly the one we are going to use.

Then we define a monotone function Φ that maps relations to relations and such that any strong bisimulation is a pre-fixpoint of Φ. However we would like to take the largest relation, not the least one, because strong bisimilarity distinguishes as least as possible. Therefore, we need a $CPO_\perp$ in which a set with more pairs is considered "smaller" than one with fewer pairs. This way, we can start from the coarsest relation, which considers all the states equivalent and, by using the approximation function, we can compute the relation that identifies only strong bisimilar agents.

We define the order relation $\sqsubseteq$ on $\wp(\mathscr{P} \times \mathscr{P})$ by letting

$$R \sqsubseteq R' \quad \Leftrightarrow \quad R' \subseteq R.$$

Notably, the bottom element is not the empty relation, but the universal relation $\mathscr{P} \times \mathscr{P}$. The resulting $CPO_\perp$ $(\mathscr{P}(P \times P), \sqsubseteq)$ is represented in Figure 11.10.

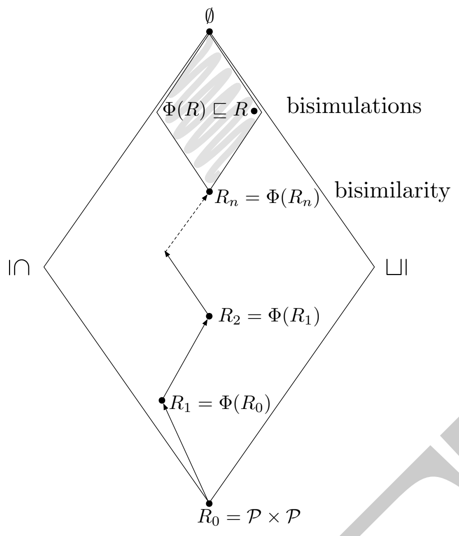


Fig. 11.10: The $CPO_\perp$ $(\wp(\mathscr{P} \times \mathscr{P}), \sqsubseteq)$

Now we define the transformation function $\Phi : \wp(\mathscr{P} \times \mathscr{P}) \to \wp(\mathscr{P} \times \mathscr{P})$.

$$p\ \Phi(R)\ q \overset{\text{def}}{=} \begin{cases} \forall \mu, p'.\ p \xrightarrow{\mu} p' \ \text{ implies } \ \exists q'.\ q \xrightarrow{\mu} q' \text{ and } p'\ R\ q' \\ \forall \mu, q'.\ q \xrightarrow{\mu} q' \ \text{ implies } \ \exists p'.\ p \xrightarrow{\mu} p' \text{ and } p'\ R\ q' \end{cases}$$

Note that Φ maps relations to relations.

Lemma 11.1 (Strong bisimulation as a pre-fixpoint). *Let R be a relation in $\wp(\mathscr{P} \times \mathscr{P})$. It is a strong bisimulation if and only if it is a pre-fixpoint of Φ, i.e., if and only if $\Phi(R) \sqsubseteq R$ (or equivalently, $R \subseteq \Phi(R)$).*

Proof. Immediate, by definition of strong bisimulation. □

It follows from Lemma 11.1 that an alternative definition of strong bisimilarity is:

$$\simeq \overset{\text{def}}{=} \bigcup_{\Phi(R) \sqsubseteq R} R.$$

Theorem 11.4. *Strong bisimilarity if the least fixpoint of Φ.*

Proof. By Theorem 11.3 it follows that strong bisimilarity is a fixpoint of Φ. Then, the thesis follows immediately by Lemma 11.1 and by the fact that strong bisimilarity is the largest strong bisimulation. □

We would like to exploit the fixpoint theorem to compute strong bisimilarity. All we need to check is that Φ is monotone and continuous.

Theorem 11.5 (Φ is monotone). *The function Φ is monotone.*

Proof. For all relations $R_1, R_2 \in \wp(\mathscr{P} \times \mathscr{P})$, we need to prove that

$$R_1 \sqsubseteq R_2 \quad \Rightarrow \quad \Phi(R_1) \sqsubseteq \Phi(R_2).$$

Assume $R_1 \sqsubseteq R_2$, i.e., $R_2 \subseteq R_1$. We want to prove that $\Phi(R_1) \sqsubseteq \Phi(R_2)$, i.e., that $\Phi(R_2) \subseteq \Phi(R_1)$. Suppose $s_1\ \Phi(R_2)\ s_2$; we want to show that $s_1\ \Phi(R_1)\ s_2$. Take μ, s_1' such that $s_1 \xrightarrow{\mu} s_1'$. Since $s_1\ \Phi(R_2)\ s_2$, there exists s_2' such that $s_2 \xrightarrow{\mu} s_2'$ and $s_1'\ R_2\ s_2'$. But since $R_2 \subseteq R_1$, we have $s_1'\ R_1\ s_2'$. Analogously for the case when $s_2 \xrightarrow{\mu} s_2'$. □

Unfortunately, the function Φ is not continuous in general, as there are pathological processes that show that the limit of the chain $\{\Phi^n(\mathscr{P} \times \mathscr{P})\}_{n\in\mathbb{N}}$ is not a strong bisimulation. As a consequence, we cannot directly apply Kleene's fixpoint theorem.

Example 11.12. To see an example of CCS processes p and q that are not strong bisimilar but that are related by all relations in the chain $\{\Phi^n(\mathscr{P} \times \mathscr{P})\}_{n\in\mathbb{N}}$, the idea is the following. For simplicity let us focus on processes that can only perform τ-transitions. Let $r \overset{\text{def}}{=} \textbf{rec}\ x.\ \tau.x$; it can only execute infinitely many τ-transitions. Now, for $n \in \mathbb{N}$, let $p_n \overset{\text{def}}{=} \underbrace{\tau.\dots\tau}_{n \text{ times}}.\textbf{nil}$ be the process that can execute n consecutive τ-transitions. Obviously r and p_n are not strongly bisimilar for any n. Then, we take as p a process that can choose between infinitely many alternatives, each choice leading to the execution of finitely many τ-transitions. Informally,

$$p = p_1 + p_2 + \dots + p_n + \dots$$

Finally, we take $q = p + r$. Clearly p and q are not strong bisimilar, because, in the bisimulation game, Alice the attacker has a winning strategy: she chooses to execute $q \xrightarrow{\tau} r$, then Bob the defender can only reply by executing a transitions of the form $p \xrightarrow{\tau} p_n$ for some $n \in \mathbb{N}$, and we know that $r \not\simeq p_n$. Of course, infinite summations are not available in the syntax of CCS. However we can define a recursive process that exhibits the same behaviour as p. Concretely, we let ϕ be a permutation such that $\phi(\alpha) = \beta$ and take $p = (p' \mid \alpha.\textbf{nil})\backslash\alpha$, where:

$$p' \overset{\text{def}}{=} \textbf{rec}\ X.\ \overline{\alpha}.\textbf{nil} + (\beta.\overline{\alpha}.\textbf{nil} \mid X[\phi])\backslash\beta$$

Now, for any $n \in \mathbb{N}$, let $\simeq_n \overset{\text{def}}{=} \Phi^n(\mathscr{P} \times \mathscr{P})$. By definition we have, that $\simeq_0 = \mathscr{P} \times \mathscr{P}$ and $\simeq_{n+1} = \Phi(\simeq_n)$ for any $n \in \mathbb{N}$. It can be proved by mathematical induction on $n \in \mathbb{N}$ that $p_n \simeq_n r$ and that for any $s \in \mathscr{P}$ it holds $s \simeq_n s$. Now we prove that $p \simeq_n q$ for any $n \in \mathbb{N}$. The proof is by mathematical induction on n. The base case follows immediately since $\simeq_0 \overset{\text{def}}{=} \mathscr{P} \times \mathscr{P}$. For the inductive case, we want to prove that $p \simeq_{n+1} q$. We observe that any transition $p \xrightarrow{\tau} p_n$ of p can be directly simulated by the corresponding move $q \xrightarrow{\tau} p_n$ of q (and vice versa). The interesting case is when we consider the transition $q \xrightarrow{\tau} r$ of q. Then, p can simulate the move by executing the transition $p \xrightarrow{\tau} p_n$, as we know that $p_n \simeq_n r$. Hence $p\ \Phi(\simeq_n)\ r$, i.e., $p \simeq_{n+1} r$.

However, if we restrict to consider (relations over) finitely branching processes, then the function Φ is continuous. Let $\mathscr{P}_f \subseteq \mathscr{P}$ denote the set of finitely branching processes.

Theorem 11.6 (Strong bisimilarity as the least fixpoint). *Let us consider only relations over finitely branching processes. Then the function Φ is continuous. Moreover, it holds:*

$$\simeq \, = \bigsqcup_{n\in\mathbb{N}} \Phi^n(\mathscr{P}_f \times \mathscr{P}_f)$$

Proof. To prove that Φ is continuous, we need to prove that for any chain $\{R_n\}_{n\in\mathbb{N}}$ of relations over finitely branching processes:

$$\Phi\left(\bigsqcup_{n\in\mathbb{N}} R_n\right) = \bigsqcup_{n\in\mathbb{N}} \Phi(R_n)$$

We prove the two inclusions separately.

$\subseteq$: Take $(p,q) \in \Phi(\bigsqcup_{n\in\mathbb{N}} R_n)$; we want to prove that $(p,q) \in \bigsqcup_{n\in\mathbb{N}} \Phi(R_n)$. This amounts to prove that $\forall n \in \mathbb{N}.\ (p,q) \in \Phi(R_n)$. Take a generic $k \in \mathbb{N}$, we want to prove that $(p,q) \in \Phi(R_k)$. Let $p \xrightarrow{\mu} p'$ of p, we want to find a transition $q \xrightarrow{\mu} q'$ of q such that $(p',q') \in R_k$. Since $(p,q) \in \Phi(\bigsqcup_{n\in\mathbb{N}} R_n)$, we know that there exists a transition $q \xrightarrow{\mu} q'$ of q such that $(p',q') \in \bigsqcup_{n\in\mathbb{N}} R_n$. Therefore $(p',q') \in R_k$. The case when q moves is analogous.

$\supseteq$: Take $(p,q) \in \bigsqcup_{n\in\mathbb{N}} \Phi(R_n)$, i.e., $\forall n \in \mathbb{N}.\ (p,q) \in \Phi(R_n)$; we want to prove that $(p,q) \in \Phi(\bigsqcup_{n\in\mathbb{N}} R_n)$. Take any transition $p \xrightarrow{\mu} p'$ of p. We want to find a transition $q \xrightarrow{\mu} q'$ of q such that $(p',q') \in \bigsqcup_{n\in\mathbb{N}} R_n$. This amounts to require that $\forall n \in \mathbb{N}.\ (p',q') \in R_n$. Since $\forall n \in \mathbb{N}.\ (p,q) \in \Phi(R_n)$, we know that for any $n \in \mathbb{N}$ there exists a transition $q \xrightarrow{\mu} q_n$ such that $(p',q_n) \in R_n$. Moreover, since $\{R_n\}_{n\in\mathbb{N}}$ is a chain, then $(p',q_n) \in R_k$ for any $k \leq n$. Since q is finitely branching, the set $\{q' \mid q \xrightarrow{\mu} q'\}$ is finite. Therefore there is some index $m \in \mathbb{N}$ such that the set $\{n \mid q_n = q_m\}$ is infinite, i.e., such that $(p,q_m) \in R_n$ for all $n \in \mathbb{N}$. We take $q' = q_m$ and we are done. The case when q moves is analogous.

The second part of the theorem, the one about $\simeq$ follows by continuity of Φ, by Kleene's fixpoint Theorem 5.6 and Theorem 11.4. $\square$

Example 11.13 (Infinitely branching process). Let us consider the recursive agent

$$p \overset{\text{def}}{=} \mathbf{rec}\ x.\ (x \mid \alpha.\,\mathbf{nil}).$$

The agent p is not guarded, because the occurrence of x in the body of the recursive process is not prefixed by an action: $G(p,\varnothing) = G(x \mid \alpha.\,\mathbf{nil},\{x\}) = G(x,\{x\}) \wedge G(\alpha.\,\mathbf{nil},\{x\}) = x \notin \{x\} \wedge G(\mathbf{nil},\varnothing) = \mathit{false} \wedge \mathit{true} = \mathit{false}$. By using the rules of the operational semantics of CCS we have, e.g.:

$$\begin{array}{rl}
\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil}) \xrightarrow{\mu} q \quad \nwarrow_{\text{Rec}} & (\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil})) \mid \alpha.\mathbf{nil} \xrightarrow{\mu} q \\
\nwarrow_{\text{Par},\ q=q_1 \mid \alpha.\mathbf{nil}} & \mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil}) \xrightarrow{\mu} q_1 \\
\nwarrow_{\text{Rec}} & (\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil})) \mid \alpha.\mathbf{nil} \xrightarrow{\mu} q_1 \\
\nwarrow_{\text{Par},\ q_1=q_2 \mid \alpha.\mathbf{nil}} & \mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil}) \xrightarrow{\mu} q_2 \\
\nwarrow_{\text{Rec}} & \cdots \\
\ldots & \mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil}) \xrightarrow{\mu} q_n \\
\nwarrow_{\text{Rec}} & (\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil})) \mid \alpha.\mathbf{nil} \xrightarrow{\mu} q_n \\
\nwarrow_{\text{Par},\ q_n=(\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil})) \mid q'} & \alpha.\mathbf{nil} \xrightarrow{\mu} q' \\
\nwarrow_{\text{Act},\ \mu=\alpha,\ q'=\mathbf{nil}} & \square
\end{array}$$

It is then evident that for any $n \in \mathbb{N}$ we have:

$$\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil}) \xrightarrow{\alpha} (\mathbf{rec}\ x.\ (x \mid \alpha.\mathbf{nil})) \mid \mathbf{nil} \mid \underbrace{\alpha.\mathbf{nil} \mid \cdots \mid \alpha.\mathbf{nil}}_{n}.$$

The problem with the processes considered in Examples 11.12 and 11.13 is that they are not guarded (see Remark 11.1), i.e. they have recursively defined names that occur *unguarded* (not nested under some action prefix) in the body of the recursive definition. The following lemma ensures that the LTS of any guarded term is finitely branching and we know already from Remark 11.1 that all states reachable from guarded processes are also guarded. As a corollary, strong bisimilarity of two guarded processes can be studied by computing the least fixpoint as in Theorem 11.6.

Lemma 11.2 (Guarded processes are finitely branching). *Let p be a guarded process. Then, for any action μ the set $\{q \mid p \xrightarrow{\mu} q\}$ is finite.*

Proof. We want to prove that $G(p,\varnothing)$ implies that the set $\{q \mid p \xrightarrow{\mu} q\}$ is finite. We prove the stronger property that for any finite set $X = \{x_1,\ldots,x_n\}$ process names and processes $p_1,\ldots,p_n$, then $G(p,X) \wedge \bigwedge_{i\in[1,n]} G(p_i,X)$ implies that $\{q \mid p[^{p_1}/_{x_1},\ldots,^{p_n}/_{x_n}] \xrightarrow{\mu} q\}$ is finite. The proof is by structural induction on p. For brevity, let σ denote the substitution $[^{p_1}/_{x_1},\ldots,^{p_n}/_{x_n}]$. We only shows a few cases.

nil:	The case where $p = \mathbf{nil}$ is trivial as $\mathbf{nil}\,\sigma = \mathbf{nil}$ and $\{q \mid \mathbf{nil} \xrightarrow{\mu} q\} = \varnothing$.
var:	If $p = x$, then there are two possibilities. If $x \in X$, then the premise $G(x,X)$ is falsified and therefore the implication holds trivially. If $x \notin X$ then $x\sigma = x$ and $\{q \mid x \xrightarrow{\mu} q\} = \varnothing$.
prefix:	If $p = \mu.p'$, then $\{q \mid (\mu.p')\sigma \xrightarrow{\mu} q\} = \{p'\sigma\}$ is a singleton.
restriction:	If $p = p'\backslash\alpha$ such that $G(p',X)$, then there are two cases. If $\mu \in \{\alpha,\overline{\alpha}\}$ then $\{q \mid (p'\backslash\alpha)\sigma \xrightarrow{\mu} q\} = \varnothing$. Otherwise the set

$$\{q \mid (p'\backslash\alpha)\sigma \xrightarrow{\mu} q\} = \{q'\backslash\alpha \mid p'\sigma \xrightarrow{\mu} q'\}$$

is finite because $\{q' \mid p'\sigma \xrightarrow{\mu} q'\}$ is finite by inductive hypothesis.

sum: If $p = p'_0 + p'_1$ such that $G(p'_0, X)$ and $G(p'_1, X)$, then the set

$$\{q \mid (p'_0 + p'_1)\sigma \xrightarrow{\mu} q\} = \{q'_0 \mid p'_0\sigma \xrightarrow{\mu} q'_0\} \cup \{q'_1 \mid p'_1\sigma \xrightarrow{\mu} q'_1\}$$

is finite because the sets $\{q'_0 \mid p'_0\sigma \xrightarrow{\mu} q'_0\}$ and $\{q'_1 \mid p'_1\sigma \xrightarrow{\mu} q'_1\}$ are finite by inductive hypothesis.

recursion: If $p = \textbf{rec}\ x.\ p'$ such that $G(p', X \cup \{x\})$,[3] then the set

$$\{q \mid (\textbf{rec}\ x.\ p')\sigma \xrightarrow{\mu} q\} = \{q \mid p'\sigma[^{\textbf{rec}\ x.\ p'}/_x] \xrightarrow{\mu} q\}$$

is finite by inductive hypothesis. □

When we want to compare two processes p and q for strong bisimilarity it is not necessary to compute the whole relation $\simeq$. Instead, we can just focus on the processes that are reachable from p and q. If the number of reachable states is finite, then the calculation is effective, but possibly quite complex if the number of states is large. In fact, the size of the LTS can explode for concise processes, due to the interleaving of concurrent actions: if we have n processes $p_1, \dots, p_n$ running in parallel, each with k possibly reachable states, then the process $((p_1 \mid p_2) \mid \dots p_n)$ can have up to k^n reachable states.

Example 11.14 (Strong bisimilarity as fixpoint). Let us consider the Example 11.9 which we have already approached with game theory techniques. Now we illustrate how to apply the fixpoint technique to the same system. Remind that:

$$p \overset{\text{def}}{=} \alpha.(\beta.\textbf{nil} + \gamma.\textbf{nil}) \qquad q \overset{\text{def}}{=} \alpha.\beta.\textbf{nil} + \alpha.\gamma.\textbf{nil}$$

Let us focus on the set of reachable states S and represent the relations by showing the equivalence classes which they induce (over reachable processes). We start with the coarsest relation, where any two processes are related (just one equivalence class). At each iteration, we refine the relation by applying the operator Φ.

$$\begin{aligned}
R_0 = \Phi^0(\bot_{\wp(S\times S)}) = \bot_{\wp(S\times S)} &= \{\ \{p\ ,\ q\ ,\ \beta.\textbf{nil} + \gamma.\textbf{nil}\ ,\ \beta.\textbf{nil}\ ,\ \gamma.\textbf{nil}\ ,\ \textbf{nil}\}\ \} \\
R_1 = \Phi(R_0) &= \{\ \{p, q\}\ ,\ \{\beta.\textbf{nil} + \gamma.\textbf{nil}\}\ ,\ \{\beta.\textbf{nil}\}\ ,\ \{\gamma.\textbf{nil}\}\ ,\ \{\textbf{nil}\}\ \} \\
R_2 = \Phi(R_1) &= \{\ \{p\}\ ,\ \{q\}\ ,\ \{\beta.\textbf{nil} + \gamma.\textbf{nil}\}\ ,\ \{\beta.\textbf{nil}\}\ ,\ \{\gamma.\textbf{nil}\}\ ,\ \{\textbf{nil}\}\ \}
\end{aligned}$$

Initially, according to R_0, any process is related with any other process, i.e., we have a unique equivalence class.

After the first iteration (R_1), we distinguish the processes on the basis of their possible transitions. Note that, as all the target states are related by R_0, we can only discriminate by looking at the labels of transitions. For example, $\beta.\textbf{nil}$ and $\gamma.\textbf{nil}$ must

[3] Without loss of generality, it can be assumed that $x \notin X$ and that x does not appear free in any p_i, as otherwise we can just α-rename x in p'. Then, for any $i \in [1, n]$ we have $G(p_i, X \cup \{x\})$ (see Remark 11.1).

be distinguished because $\beta.\,\mathbf{nil}$ has an outgoing β-transition, while $\gamma.\,\mathbf{nil}$ does not have a β-transition. Similarly $\beta.\,\mathbf{nil}+\gamma.\,\mathbf{nil}$ must be distinguished from $\gamma.\,\mathbf{nil}$ because it has a β-transition and from $\beta.\,\mathbf{nil}$ because it has a γ-transition. Moreover, the inactive process **nil** is clearly distinguished from any other (non deadlock) process. Only p and q are related by R_1, because both can execute only α-transitions.

At the second iteration we focus on the unique equivalence class $\{p,q\}$ in R_1 which is not a singleton, as we cannot split any further the other equivalence classes. Now let us consider the transition $q \xrightarrow{\alpha} \beta.\,\mathbf{nil}$. Process p has a unique α-transition that can be used to simulate the move of q, namely $p \xrightarrow{\alpha} \beta.\,\mathbf{nil}+\gamma.\,\mathbf{nil}$, but $\beta.\,\mathbf{nil}$ and $\beta.\,\mathbf{nil}+\gamma.\,\mathbf{nil}$ are not related by R_1, therefore p and q must be distinguished by R_2.

Note that R_2 is a fixpoint, because each equivalence class is a singleton and cannot be split any further. Hence p and q fall in different equivalence classes and they are not strong bisimilar.

We conclude by studying strong bisimilarity of possibly unguarded processes. Even in this case the least fixpoint exists, as granted by Knaster-Tarski's fixpoint Theorem 11.7 which ensures the existence of least and greatest fixpoints for monotone functions over *complete lattices*.

Definition 11.8 (Complete lattice). A partial order $(D,\sqsubseteq)$ is a *complete lattice* if any subset $X \subseteq D$ has a least upper bound and a greatest lower bound, denoted by $\bigsqcup X$ and $\bigsqcap X$, respectively.

Note that any complete lattice has a least element $\bot = \bigsqcap D$ and a greatest element $\top = \bigsqcup D$. Any powerset ordered by inclusion defines a complete lattice, hence the set $\wp(\mathscr{P} \times \mathscr{P})$ of all relations over CCS processes is a complete lattice.

The next important result is named after Bronislaw Knaster who proved it for the special case of lattices of sets and Alfred Tarski who generalised the theorem to its current formulation.[4]

Theorem 11.7 (Knaster-Tarski's fixpoint theorem). *Let $(D,\sqsubseteq)$ a complete lattice and $f : D \to D$ a monotone function. Then f has a least fixpoint and a greatest fixpoint, defined respectively as follows:*

$$d_{min} \stackrel{\text{def}}{=} \bigsqcap \{d \in D \mid f(d) \sqsubseteq d\} \qquad d_{max} \stackrel{\text{def}}{=} \bigsqcup \{d \in D \mid d \sqsubseteq f(d)\}.$$

Proof. It can be seen that d_{min} is defined as the greatest lower bound of the set of pre-fixpoints. To prove that d_{min} is the least fixpoint, we need to prove that:

1. d_{min} is a fixpoint, i.e., $f(d_{min}) = d_{min}$;
2. for any other fixpoint $d \in D$ of f we have $d_{min} \sqsubseteq d$.

We split the proof of point 1, in two parts: $f(d_{min}) \sqsubseteq d_{min}$ and $d_{min} \sqsubseteq f(d_{min})$.

For conciseness, let $Pre_f \stackrel{\text{def}}{=} \{d \in D \mid f(d) \sqsubseteq d\}$. By definition of d_{min}, we have $d_{min} \sqsubseteq d$ for any $d \in Pre_f$. Since f is monotone, $f(d_{min}) \sqsubseteq f(d)$ and by transitivty

[4] The theorem is actually stronger than what is presented here, because it asserts that the set of fixpoints of a monotone function on a complete lattice forms a complete lattice itself.

$$f(d_{min}) \sqsubseteq f(d) \sqsubseteq d$$

Thus, also $f(d_{min})$ is a lower bound of the set $\{d \in D \mid f(d) \sqsubseteq d\}$. Since d_{min} is the greatest lower bound, we have $f(d_{min}) \sqsubseteq d_{min}$.

To prove the converse, note that by the previous property and monotonicity of f we have $f(f(d_{min})) \sqsubseteq f(d_{min})$. Therefore $f(d_{min}) \in Pre_f$ and since d_{min} is a lower bound of Pre_f it must be $d_{min} \sqsubseteq f(d_{min})$.

Finally, any fixpoint $d \in D$ of f is also a pre-fixpoint, i.e., $d \in Pre_f$ and thus $d_{min} \sqsubseteq d$ because d_{min} is a lower bound of Pre_f.

The proof that d_{max} is the greatest fixpoint is analogous and thus omitted. □

We have already seen that Φ is monotone, hence Knaster-Tarski's fixpoint theorem guarantees the existence of the least fixpoint, and hence strong bisimilarity, also when infinitely branching processes are considered.

11.5 Compositionality

In this section we focus on *compositionality* issues of the abstract semantics which we have just introduced. For an abstract semantics to be practically relevant it is important that any process used in a system can be replaced with an equivalent process without changing the semantics of the system. Since we have not used structural induction in defining the abstract semantics of CCS, no kind of compositionality is ensured w.r.t. the possible ways of constructing larger systems, i.e., w.r.t. the operators of CCS.

Definition 11.9 (Congruence). An equivalence $\equiv$ is said to be a *congruence* (with respect to a class of contexts) if:

$$\forall C[\cdot]. \quad p \equiv q \quad \Rightarrow \quad C[p] \equiv C[q]$$

In order to guarantee the compositionality of CCS we must show that strong bisimilarity is a congruence relation.

Let us now see an example of a relation which is not a congruence.

Example 11.15 (Completed trace semantics). Let us consider the processes p and q from Example 11.9. Take the following context:

$$C[\cdot] \stackrel{\text{def}}{=} (\,\cdot \mid \overline{\alpha}.\overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

Now we can fill the hole in $C[\cdot]$ with the processes p and q:

$$C[p] = (\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \mid \overline{\alpha}.\overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

$$C[q] = ((\alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}) \mid \overline{\alpha}.\overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

Obviously $C[p]$ and $C[q]$ generate the same set of traces, however one of the processes can "deadlock" before the interaction on β takes place, but not the other:

$$\begin{tikzcd}
C[p] \arrow[d, "\tau"] & C[q] \arrow[d, "\tau"'] \arrow[dd, bend left, "\tau"] \\
((\beta.\,\mathbf{nil}+\gamma.\,\mathbf{nil}) \mid \overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma \arrow[d, "\tau"] & (\gamma.\,\mathbf{nil} \mid \overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma \\
(\mathbf{nil} \mid \overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma \arrow[d, "\overline{\delta}"] & (\beta.\,\mathbf{nil} \mid \overline{\beta}.\overline{\delta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma \arrow[l, "\tau"'] \\
(\mathbf{nil} \mid \mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma &
\end{tikzcd}$$

The difference can be formalised if we consider the so-called *completed trace semantics*. Let us write $p \not\rightarrow$ for the predicate $\neg(\exists\mu.\ \exists q.\ p \xrightarrow{\mu} q)$. A *completed trace* of a process p is a sequence of actions $\mu_1 \cdots \mu_k$ (for $k \geq 0$) such that there exists a sequence of transitions

$$p = p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_{k-1}} p_{k-1} \xrightarrow{\mu_k} p_k \not\rightarrow$$

for some $p_1, \ldots, p_k$. The completed traces of a process characterise the sequences of actions that can lead the system to a deadlocked configuration, where no further action is possible.

The completed trace semantics of p is the same as that of q, namely $\{\ \alpha\beta\ ,\ \alpha\gamma\ \}$. However, the completed traces of $C[p]$ and $C[q]$ are $\{\ \tau\tau\overline{\delta}\ \}$ and $\{\ \tau\tau\overline{\delta}\ ,\ \tau\ \}$, respectively. We can thus conclude that the completed trace semantics is not a congruence.

11.5.1 Strong bisimilarity is a Congruence

In order to show that strong bisimilarity is a congruence w.r.t. all contexts it is enough to prove that the property holds for all the operators of CCS. So we need to prove that, for any $p, p_0, p_1, q, q_0, q_1 \in \mathscr{P}$:

- if $p \simeq q$, then $\forall\mu.\ \mu.p \simeq \mu.q$;
- if $p \simeq q$, then $\forall\alpha.\ p\backslash\alpha \simeq q\backslash\alpha$;
- if $p \simeq q$, then $\forall\phi.\ p[\phi] \simeq q[\phi]$;
- if $p_0 \simeq q_0$ and $p_1 \simeq q_1$, then $p_0 + p_1 \simeq q_0 + q_1$;
- if $p_0 \simeq q_0$ and $p_1 \simeq q_1$, then $p_0 \mid p_1 \simeq q_0 \mid q_1$.

The congruence property is important, because it allows to replace any process with an equivalent one in any context preserving the overall behaviour.

Here we give the proof only for parallel composition, which is an interesting case to consider. The other cases follow by similar arguments and are left as an exercise (see Problem 11.7)

Lemma 11.3 (Strong bisimilarity is preserved by parallel composition). *For any $p_0, p_1, q_0, q_1 \in \mathscr{P}$, if $p_0 \simeq q_0$ and $p_1 \simeq q_1$, then $p_0 \mid p_1 \simeq q_0 \mid q_1$.*

Proof. As usual we assume the premise $p_0 \simeq q_0 \wedge p_1 \simeq q_1$ and we would like to prove that $p_0 \mid p_1 \simeq q_0 \mid q_1$, i.e., that:

$$\exists R.\ (p_0 \mid p_1)\, R\, (q_0 \mid q_1)\ \wedge\ R \subseteq \Phi(R)$$

Since $p_0 \simeq q_0$ and $p_1 \simeq q_1$ we have:

$$\begin{array}{ll} p_0\, R_0\, q_0 & \text{for some strong bisimulation } R_0 \subseteq \Phi(R_0) \\ p_1\, R_1\, q_1 & \text{for some strong bisimulation } R_1 \subseteq \Phi(R_1) \end{array}$$

Now let us consider the relation:

$$R \stackrel{\text{def}}{=} \{(r_0 \mid r_1\ ,\ s_0 \mid s_1)\ \mid\ r_0\, R_0\, s_0\ \wedge\ r_1\, R_1\, s_1\}$$

By definition it holds $(p_0 \mid p_1)\, R\, (q_0 \mid q_1)$. Now we show that R is a strong bisimulation (i.e., that $R \subseteq \Phi(R)$). Let us take a generic pair $(r_0 \mid r_1\ ,\ s_0 \mid s_1) \in R$ and let us consider a transition $r_0 \mid r_1 \xrightarrow{\mu} r$, we need to prove that there exists s such that $s_0 \mid s_1 \xrightarrow{\mu} s$ with $(r,s) \in R$. (The case where $s_0 \mid s_1$ executes a transition that $r_0 \mid r_1$ must simulate is completely analogous.) There are three rules whose conclusions have the form $r_0 \mid r_1 \xrightarrow{\mu} r$.

- The first case is when we have applied the first (Par) rule. So we have $r_0 \xrightarrow{\mu} r_0'$ and $r = r_0' \mid r_1$ for some r_0'. Since $r_0\, R_0\, s_0$ and R_0 is a strong bisimulation relation, then there exists s_0' such that $s_0 \xrightarrow{\mu} s_0'$ and $(r_0', s_0') \in R_0$. Then, by applying the same inference rule we get $s_0 \mid s_1 \xrightarrow{\mu} s_0' \mid s_1$. Since $(r_0', s_0') \in R_0$ and $(r_1, s_1) \in R_1$, we have $(r_0' \mid r_1, s_0' \mid s_1) \in R$ and we conclude by taking $s = s_0' \mid s_1$.
- The second case is when we have applied the second (Par) rule. So we have $r_1 \xrightarrow{\mu} r_1'$ and $r = r_0 \mid r_1'$ for some r_1'. By a similar argument to the previous case we prove the thesis.
- The last case is when we have applied the (Com) rule. This means that $r_0 \xrightarrow{\lambda} r_0'$, $r_1 \xrightarrow{\overline{\lambda}} r_1'$, $\mu = \tau$ and $r = r_0' \mid r_1'$ for some observable action λ and processes r_0', r_1'. Since $r_0\, R_0\, s_0$ and R_0 is a strong bisimulation relation, then there exists s_0' such that $s_0 \xrightarrow{\lambda} s_0'$ and $(r_0', s_0') \in R_0$. Similarly, since $r_1\, R_1\, s_1$ and R_1 is a strong bisimulation relation, then there exists s_1' such that $s_1 \xrightarrow{\overline{\lambda}} s_1'$ and $(r_1', s_1') \in R_1$. Then, by applying the same inference rule we get $s_0 \mid s_1 \xrightarrow{\tau} s_0' \mid s_1'$. Since $(r_0', s_0') \in R_0$ and $(r_1', s_1') \in R_1$, we have $(r_0' \mid r_1', s_0' \mid s_1') \in R$ and we conclude by taking $s = s_0' \mid s_1'$. $\square$

11.6 A Logical View to Bisimilarity: Hennessy-Milner Logic

In this section we present a *modal logic* introduced by Matthew Hennessy and Robin Milner. Modal logic allows to express concepts as "there exists a next state such that", or "for all next states", some property holds. Typically, model checkable properties are stated as formulas in some modal logic. In particular, Hennessy-Milner modal logic is relevant for its simplicity and for its close connection to strong bisimilarity. As we will see, in fact, two strong bisimilar agents satisfy the same set of modal logic formulas. This fact shows that strong bisimilarity is at the right level of abstraction.

Definition 11.10. The formulas of *Hennessy-Milner logic* (HM-logic) are generated by the following grammar:

$$F \quad ::= \quad \mathit{true} \quad | \quad \mathit{false} \quad | \quad \bigwedge_{i\in I} F_i \quad | \quad \bigvee_{i\in I} F_i \quad | \quad \Diamond_\mu F \quad | \quad \Box_\mu F$$

We write $\mathscr{L}$ for the set of the HM-logic formulas (*HM-formulas* for short).

The formulas of HM-logic express properties over the states of an LTS, i.e., in our case, of CCS agents. The meanings of the logic operators are the following:

true: is the formula satisfied by every agent. This operator is sometimes written *tt* or just T.

false: is the formula never satisfied by any agent. This operator is sometimes written *ff* or just F.

$\bigwedge_{i\in I} F_i$: corresponds to the conjunction of the formulas in $\{F_i\}_{i\in I}$. Notice that *true* can be considered as a shorthand for an indexed conjunction where the set I of indexes is empty.

$\bigvee_{i\in I} F_i$: corresponds to the disjunction of the formulas in $\{F_i\}_{i\in I}$. Notice that *false* can be considered as a shorthand for an indexed disjunction where the set I of indexes is empty.

$\Diamond_\mu F$: it is a *modal operator*; an agent p satisfies this formula if there exists a μ-labelled transition from p to some state q that satisfies and the formula F. This operator is sometimes written $\langle\mu\rangle F$.

$\Box_\mu F$: it is a *modal operator*; an agent p satisfies this formula if for any q such that there is a μ-labelled transition from p to q the formula F is satisfied by q. This operator is sometimes written $[\mu]F$.

As usual, logical satisfaction is defined as a relation $\models$ between formulas and their models, which in our case are CCS processes, seen as states of the LTS defined by the operational semantics.

Definition 11.11 (Satisfaction relation). The *satisfaction relation* $\models\, \subseteq \mathscr{P} \times \mathscr{L}$ is defined as follows (for any $p \in \mathscr{P}$, $F \in \mathscr{L}$ and $\{F_i\}_{i\in I} \subseteq \mathscr{L}$):

$$\begin{array}{lll}
p \models \mathit{true} & & \\
p \models \bigwedge_{i\in I} F_i & \text{iff} & \forall i \in I.\ p \models F_i \\
p \models \bigvee_{i\in I} F_i & \text{iff} & \exists i \in I.\ p \models F_i \\
p \models \Diamond_\mu F & \text{iff} & \exists p'.\ p \xrightarrow{\mu} p' \wedge p' \models F \\
p \models \Box_\mu F & \text{iff} & \forall p'.\ p \xrightarrow{\mu} p' \Rightarrow p' \models F
\end{array}$$

If $p \models F$ we say that the process p *satisfies* the HM-formula F.

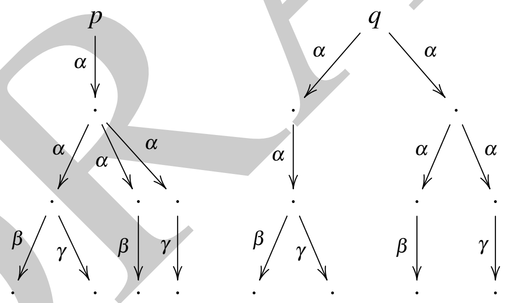
Notably, if p cannot execute any μ-transition, then $p \models \Box_\mu F$ for any formula F. For example, the formula $\Diamond_\alpha \mathit{true}$ is satisfied by all processes that can execute an α-transition, and the formula $\Box_\beta \mathit{false}$ is satisfied by all processes that cannot execute a β-transition. Then the formula $\Diamond_\alpha \mathit{true} \wedge \Box_\beta \mathit{false}$ is satisfied by all processes that can execute an α-transition but not a β-transition, while the formula $\Diamond_\alpha \Box_\beta \mathit{false}$ is satisfied by all processes that can execute an α-transition to reach a state where no β-transition can be executed. Can you guess by which processes are satisfied the formulas $\Diamond_\alpha \mathit{false}$ and $\Box_\beta \mathit{true}$? And the formula $\Box_\beta \Diamond_\alpha \mathit{true}$?

HM-logic induces an obvious equivalence on CCS processes: Two agents are logically equivalent if they satisfy the same set of formulas.

Definition 11.12 (HM-logic equivalence). Let p and q be two CCS processes. We say that p and q are *HM-logic equivalent*, written $p \equiv_{HM} q$ if

$$\forall F \in \mathscr{L}. \quad p \models F \quad \Leftrightarrow \quad q \models F.$$

Example 11.16 (Non-equivalent agents). Let us consider two CCS agents p and q whose LTSs are below:



We would like to show a formula F which is satisfied by one of the two agents and not by the other. For example, if we take

$$F = \Diamond_\alpha \Box_\alpha (\Diamond_\beta \mathit{true} \wedge \Diamond_\gamma \mathit{true}) \qquad \text{we have} \qquad p \not\models F \qquad \text{and} \qquad q \models F.$$

The agent p does not satisfy the formula F because after having executed its unique α-transition we reach a state where it is possible to take α-transitions that lead to states where either β or γ is enabled, but not both. On the contrary, we can execute the leftmost α-transition of q and we reach a state that satisfies $\Box_\alpha (\Diamond_\beta \mathit{true} \wedge \Diamond_\gamma \mathit{true})$ (i.e., the (only) state reachable by an α-transition can perform both γ and β).

Although negation is not present in the syntax, HM-logic is closed under negation, i.e., taken any formula F we can easily compute another formula F^{c} such that

$$\forall p \in \mathscr{P}.\ p \models F \Leftrightarrow p \not\models F^{\mathrm{c}}.$$

The converse formula F^{c} is defined by structural recursion as follows:

$$\begin{array}{rlrl} \mathit{true}^{\mathrm{c}} & \stackrel{\mathrm{def}}{=} \mathit{false} & \mathit{false}^{\mathrm{c}} & \stackrel{\mathrm{def}}{=} \mathit{true} \\ (\bigwedge_{i\in I} F_i)^{\mathrm{c}} & \stackrel{\mathrm{def}}{=} \bigvee_{i\in I} F_i^{\mathrm{c}} & (\bigvee_{i\in I} F_i)^{\mathrm{c}} & \stackrel{\mathrm{def}}{=} \bigwedge_{i\in I} F_i^{\mathrm{c}} \\ (\Diamond_\mu F)^{\mathrm{c}} & \stackrel{\mathrm{def}}{=} \Box_\mu F^{\mathrm{c}} & (\Box_\mu F)^{\mathrm{c}} & \stackrel{\mathrm{def}}{=} \Diamond_\mu F^{\mathrm{c}} \end{array}$$

Now we present two theorems which allow us to connect strong bisimilarity and modal logic. As we said this connection is very important both from theoretical and practical point of view. We start by introducing a measure over formulas, called *modal depth*, to estimate the maximal number of consecutive steps that must be taken into account to check the validity of the formulas.

Definition 11.13 (Depth of a formula). We define the *modal depth* (also *depth*) of a formula as follows:

$$\begin{aligned} \mathrm{md}(\mathit{true}) = \mathrm{md}(\mathit{false}) &\stackrel{\mathrm{def}}{=} 0 \\ \mathrm{md}(\bigwedge_{i\in I} F_i) = \mathrm{md}(\bigvee_{i\in I} F_i) &\stackrel{\mathrm{def}}{=} \max\{\mathrm{md}(F_i) \mid i \in I\} \\ \mathrm{md}(\Diamond_\mu F) = \mathrm{md}(\Box_\mu F) &\stackrel{\mathrm{def}}{=} 1 + \mathrm{md}(F) \end{aligned}$$

It is immediate to see that the modal depth corresponds to the maximum nesting level of modal operators. Moreover $\mathrm{md}(F^{\mathrm{c}}) = \mathrm{md}(F)$ (see Problem 11.16). For example, in the case of the formula F in Example 11.16, we have $\mathrm{md}(F) = 3$. We will denote the set of logic formulas of modal depth k with $\mathscr{L}_k = \{F \in \mathscr{L} \mid \mathrm{md}(F) = k\}$.

The first theorem ensures that if two agents are not distinguished by the k-th iteration of the fixpoint calculation of strong bisimilarity, then no formula of depth k can distinguish between the two agents, and vice versa.

Theorem 11.8. *Let $k \in \mathbb{N}$ and let the relation $\simeq_k$ be defined as follows (see Example 11.12):*

$$p \simeq_k q \quad \Leftrightarrow \quad p\, \Phi^k(\mathscr{P}_f \times \mathscr{P}_f)\, q.$$

Then, we have:

$$\forall k \in \mathbb{N}.\ \forall p, q \in \mathscr{P}_f.\ p \simeq_k q \quad \mathit{iff} \quad \forall F \in \mathscr{L}_k.\ (p \models F) \Leftrightarrow (q \models F).$$

Proof. We proceed by strong mathematical induction on k.

Base case: for $k = 0$ the only formulas F with $\mathrm{md}(F) = 0$ are (conjunctions and disjunctions of) *true* and *false*, which cannot be used to distinguish processes. In fact $\Phi^0(\mathscr{P}_f \times \mathscr{P}_f) = \mathscr{P}_f \times \mathscr{P}_f$.

Ind. case: Suppose that:

$$\forall p, q \in \mathscr{P}_f.\ p \simeq_k q \quad \text{iff} \quad \forall F \in \mathscr{L}_k.\ (p \models F) \Leftrightarrow (q \models F).$$

We want to prove that

$$\forall p, q \in \mathscr{P}_f.\ p \simeq_{k+1} q \quad \text{iff} \quad \forall F \in \mathscr{L}_{k+1}.\ (p \models F) \Leftrightarrow (q \models F).$$

We prove that

1. If $p \not\simeq_{k+1} q$ then a formula $F \in \mathscr{L}_{k+1}$ can be found such that $p \models F$ and $q \not\models F$. Without loss of generality, suppose there are μ, p' such that $p \xrightarrow{\mu} p'$ and for any q' such that $q \xrightarrow{\mu} q'$ then $p' \not\simeq_k q'$. By inductive hypothesis, for any q' such that $q \xrightarrow{\mu} q'$ there exists a formula $F_{q'} \in \mathscr{L}_k$ that is satisfied[5] by p' and not by q'. Since q is finitely branching, the set $Q \stackrel{\text{def}}{=} \{q' \mid q \xrightarrow{\mu} q'\}$ is finite and we can set
$$F \stackrel{\text{def}}{=} \Diamond_\mu \bigwedge_{q' \in Q} F_{q'}.$$
2. If $p \simeq_{k+1} q$ and $p \models F$ then $q \models F$. The proof proceeds by structural induction on F. We leave the reader to fill the details. □

The second theorem generalises the above correspondence by setting up a connection between formulas of any depth and strong bisimilarity.

Theorem 11.9. *Let p and q two finitely branching CCS processes, then we have:*

$$p \simeq q \quad \textit{if and only if} \quad p \equiv_{HM} q$$

Proof. It is a consequence of Theorems 11.6 and 11.8. □

It is worth reading this result both in the positive sense, namely strong bisimilar agents satisfy the same set of HM-formulas; and in the negative sense, namely if two finitely branching agents p and q are not strong bisimilar, then there exists a formula F which distinguishes between them, i.e., such that $p \models F$ but $q \not\models F$. From a theoretical point of view these theorems show that strong bisimilarity distinguishes all and only those agents which enjoy different properties. These results witness that the relation $\simeq$ is a good choice from the logical point of view. From the point of view of verification, if we are given a specification $F \in \mathscr{L}$ and a (finitely branching) implementation p, it can be convenient to minimise the size of the LTS of p by taking its quotient q up to bisimilarity and then check if $q \models F$.

Later, in Section 12.2, we will show that we can define a denotational semantics for logic formulas, by assigning to each formula F the set $\{p \mid p \models F\}$ of all processes that satisfy F.

[5] If the converse applies, we just take $F^c_{q'}$.

11.7 Axioms for Strong Bisimilarity

Finally, we show that strong bisimilarity can be finitely axiomatised. First we present a theorem which allows to derive for every non recursive CCS agent a suitable normal form.

Theorem 11.10. *Let p be a (non-recursive) CCS agent, then there exists a CCS agent, strong bisimilar to p, built using only prefix, sum and* **nil**.

Proof. We proceed by structural recursion. First we define two auxiliary binary operators $\lfloor$ and $\|$, where $p\lfloor q$ means that p must make a transition while q stays idle, and $p_1\|p_2$ means that p_1 and p_2 must perform a synchronisation. In both case, after the transition, the processes run in parallel. This corresponds to say that the operational semantics rules for $p\lfloor q$ and $p\|q$ are:

$$\frac{p \xrightarrow{\mu} p'}{p\lfloor q \xrightarrow{\mu} p' \mid q} \qquad \frac{p \xrightarrow{\lambda} p' \quad q \xrightarrow{\overline{\lambda}} q'}{p\|q \xrightarrow{\tau} p' \mid q'}$$

We show how to decompose the parallel operator, then we show how to simplify the other cases:

$$p_1 \mid p_2 \simeq p_1\lfloor p_2 \;+\; p_2\lfloor p_1 \;+\; p_1\|p_2$$

$$\begin{aligned}
\mathbf{nil}\lfloor p &\simeq \mathbf{nil}\\
\mu.p\lfloor q &\simeq \mu.(p \mid q)\\
(p_1+p_2)\lfloor q &\simeq p_1\lfloor q \;+\; p_2\lfloor q
\end{aligned}$$

$$\begin{aligned}
\mathbf{nil}\|p \simeq p\|\mathbf{nil} &\simeq \mathbf{nil}\\
\mu_1.p_1\|\mu_2.p_2 &\simeq \mathbf{nil} \;\text{ if }\; \mu_1 \neq \overline{\mu}_2 \vee \mu_1 = \tau\\
\lambda.p_1\|\overline{\lambda}.p_2 &\simeq \tau.(p_1 \mid p_2)\\
(p_1+p_2)\|q &\simeq p_1\|q \;+\; p_2\|q\\
p\|(q_1+q_2) &\simeq p\|q_1 \;+\; p\|q_2
\end{aligned}$$

$$\begin{aligned}
\mathbf{nil}\backslash\alpha &\simeq \mathbf{nil}\\
(\mu.p)\backslash\alpha &\simeq \mathbf{nil} \;\text{ if }\; \mu \in \{\alpha,\overline{\alpha}\}\\
(\mu.p)\backslash\alpha &\simeq \mu.(p\backslash\alpha) \;\text{ if }\; \mu \neq \alpha,\overline{\alpha}\\
(p_1+p_2)\backslash\alpha &\simeq p_1\backslash\alpha \;+\; p_2\backslash\alpha
\end{aligned}$$

$$\begin{aligned}
\mathbf{nil}[\phi] &\simeq \mathbf{nil}\\
(\mu.p)[\phi] &\simeq \phi(\mu).p[\phi]\\
(p_1+p_2)[\phi] &\simeq p_1[\phi] \;+\; p_2[\phi]
\end{aligned}$$

By repeatedly applying the axioms from left to right it is evident that any (non-recursive) agent p can be rewritten to a sequential agent q built using only action prefix, sum and **nil**. Since the left hand side and the right hand side of each axiom can be proved to be strong bisimilar, by transitivity and congruence of strong bisimilarity, we have that p and q are strong bisimilar. $\square$

From the previous theorem, it follows that every non-recursive CCS agent can be equivalently written using action prefix, sum and **nil**. Note that the LTS of any non-recursive CCS agent has only a finite number of reachable states. We call *finite* any such agent.

Then, the axioms that characterise the strong bisimilarity relation are the following:

$$\begin{aligned} p + \mathbf{nil} &\simeq p \\ p_1 + p_2 &\simeq p_2 + p_1 \\ p_1 + (p_2 + p_3) &\simeq (p_1 + p_2) + p_3 \\ p + p &\simeq p \end{aligned}$$

This last set of axioms simply asserts that processes with sum define an idempotent, commutative monoid whose neutral element is **nil**.

Theorem 11.11. *Any two finite CCS processes p and q are strong bisimilar if and only if they can be equated using the above axioms.*

Proof. We need to prove that the axioms are sound (i.e., they preserve strong bisimilarity) and complete (i.e., any strong bisimilar finite agents can be proved equivalent using the axioms). Soundness can be proved by showing that the left-hand side and the right-hand side of each axiom are strong bisimilar, which can be readily done by exhibiting suitable strong bisimulation relations, similarly to what has been done for proving that strong bisimilarity is a congruence. Completeness is more involved. First, it requires the definition of a normal form representation for processes, called *head normal form* (*HNF* for short). Second, it requires proving that for any two strong bisimilar processes p and q that are in HNF we can prove that p is equal to q by using the axioms. Third, it requires proving that any process can be put in HNF. Formally, a process p is in HNF if it is written $p = \sum_{i \in I} \mu_i . p_i$ for some processes p_i that are themselves in HNF. We omit here the details of the proof. $\square$

Example 11.17 (Proving strong bisimilarity by equational reasoning). We have seen in Example 11.7 that the operational semantics reduces concurrency to non-determinism. Let us prove that $\alpha.\,\mathbf{nil} \mid \beta.\,\mathbf{nil}$ is strongly bisimilar to $\alpha.\beta.\,\mathbf{nil} + \beta.\alpha.\,\mathbf{nil}$ by using the axioms for strong bisimilarity. First let us observe that

$$\mathbf{nil} \mid \mathbf{nil} \quad\simeq\quad \mathbf{nil} \lfloor \mathbf{nil} + \mathbf{nil} \lfloor \mathbf{nil} + \mathbf{nil} \,\|\, \mathbf{nil} \quad\simeq\quad \mathbf{nil} + \mathbf{nil} + \mathbf{nil} \quad\simeq\quad \mathbf{nil}$$

Then, we have

$$\begin{aligned}
\alpha.\mathbf{nil} \mid \beta.\mathbf{nil} &\simeq \alpha.\mathbf{nil}\lfloor\beta.\mathbf{nil} + \beta.\mathbf{nil}\lfloor\alpha.\mathbf{nil} + \alpha.\mathbf{nil}\|\beta.\mathbf{nil}\\
&\simeq \alpha.(\mathbf{nil} \mid \beta.\mathbf{nil}) + \beta.(\mathbf{nil} \mid \alpha.\mathbf{nil}) + \mathbf{nil}\\
&\simeq \alpha.(\mathbf{nil}\lfloor\beta.\mathbf{nil} + \beta.\mathbf{nil}\lfloor\mathbf{nil} + \mathbf{nil}\|\beta.\mathbf{nil}) +\\
&\quad\ \beta.(\mathbf{nil}\lfloor\alpha.\mathbf{nil} + \alpha.\mathbf{nil}\lfloor\mathbf{nil} + \mathbf{nil}\|\alpha.\mathbf{nil})\\
&\simeq \alpha.(\mathbf{nil} + \beta.(\mathbf{nil} \mid \mathbf{nil}) + \mathbf{nil}) + \beta.(\mathbf{nil} + \alpha.(\mathbf{nil} \mid \mathbf{nil}) + \mathbf{nil})\\
&\simeq \alpha.(\beta.(\mathbf{nil})) + \beta.(\alpha.(\mathbf{nil}))\\
&\simeq \alpha.\beta.\mathbf{nil} + \beta.\alpha.\mathbf{nil}
\end{aligned}$$

We remark that strong bisimilarity of (possibly recursive) CCS processes is not decidable in general, while the above theorem can be used to prove that strong bisimilarity of finite CCS process is decidable. Moreover, if two finitely branching (but possibly infinite-state) processes are not strong bisimilar, then we should be able to find a finite counterexample, i.e., strong bisimilarity inequivalence of finitely branching processes is semi-decidable (as a consequence of Theorem 11.9).

11.8 Weak Semantics of CCS

Let us now see an example that illustrates the limits of strong bisimilarity as a behavioural equivalence between agents.

Example 11.18 (Linked buffers). Let us consider the buffers implemented as in Example 11.8. An alternative implementation of a buffer of capacity two could be obtained by linking two buffers of capacity one. Let us define the linking operation, similarly to what we have done in Example 11.3, as follows:

$$p \frown q \stackrel{\text{def}}{=} (p[\phi_{out}] \mid q[\phi_{in}])\backslash\ell$$

where $\phi_{out}(out) = \ell$ and $\phi_{in}(in) = \ell$ and they are the identity otherwise. Then an empty buffer of capacity two could be implemented by taking $B^1_0 \frown B^1_0$. However, its LTS is

$$\begin{tikzcd}
 & B^1_0 \frown B^1_0 \arrow[dl, "in"'] & \\
B^1_1 \frown B^1_0 \arrow[rr, "\tau"] & & B^1_0 \frown B^1_1 \arrow[ul, "out"'] \arrow[dl, "in"] \\
 & B^1_1 \frown B^1_1 \arrow[ul, "out"] &
\end{tikzcd}$$

Obviously the internal τ-transition $B^1_1 \frown B^1_0 \xrightarrow{\tau} B^1_0 \frown B^1_1$, which is necessary to shift the data from the leftmost buffer to the rightmost buffer, makes it not possible to establish a strong bisimulation between B^2_0 and $B^1_0 \frown B^1_0$.

The above example shows that, when we consider τ as an internal action, not visible from outside of the system, we would like, accordingly, relate observable behaviours that differs just for τ-actions. Therefore strong bisimilarity is not abstract enough for some purposes. For example, in many situations, one can use CCS to give an abstract specification of a system and also to define an implementation that should be provable "equivalent" to the specification, but typically the implementation makes use of auxiliary invisible actions τ that are not present in the specification. So it is natural to try to abstract away from the invisible (τ-labelled) transitions by defining a new equivalence relation. This relation is called *weak bisimilarity*. We start by defining a new, more abstract, LTS, where a single transitions can involve several internal moves.

11.8.1 Weak Bisimilarity

Definition 11.14 (Weak transitions). We let $\Rightarrow$ be the *weak transition relation* on the set of states of an LTS defined as follows:

$$p \overset{\tau}{\Rightarrow} q \overset{\text{def}}{=} p \xrightarrow{\tau} \dots \xrightarrow{\tau} q \lor p = q$$
$$p \overset{\lambda}{\Rightarrow} q \overset{\text{def}}{=} \exists p', q'.\ p \overset{\tau}{\Rightarrow} p' \xrightarrow{\lambda} q' \overset{\tau}{\Rightarrow} q$$

Note that $p \overset{\tau}{\Rightarrow} q$ means that q can be reached from p via a, possibly empty, finite sequence of τ-transitions, i.e., the weak transition relation $\overset{\tau}{\Rightarrow}$ coincides with the reflexive and transitive closure $(\xrightarrow{\tau})^*$ of the silent transition relation $\xrightarrow{\tau}$. For λ an observable action, the relation $\overset{\lambda}{\Rightarrow}$ requires instead the execution of exactly one λ-transition, possibly preceded and followed by any finite sequence (also empty) of silent transitions.

We can now define a notion of bisimulation that is based on weak transitions.

Definition 11.15 (Weak Bisimulation). Let R be a binary relation on the set of states of an LTS then it is a *weak bisimulation* if

$$\forall s_1, s_2.\ s_1\ R\ s_2 \Rightarrow \begin{cases} \forall \mu, s_1'.\ \text{ if } s_1 \xrightarrow{\mu} s_1' \text{ then } \exists s_2' \text{ such that } s_2 \overset{\mu}{\Rightarrow} s_2' \text{ and } s_1'\ R\ s_2' \\ \forall \mu, s_2'.\ \text{ if } s_2 \xrightarrow{\mu} s_2' \text{ then } \exists s_1' \text{ such that } s_1 \overset{\mu}{\Rightarrow} s_1' \text{ and } s_1'\ R\ s_2' \end{cases}$$

Definition 11.16 (Weak bisimilarity $\approx$). Let s_1 and s_2 be two states of an LTS, then they are said to be *weak bisimilar*, written $s_1 \approx s_2$ if there exists a weak bisimulation R such that $s_1\ R\ s_2$.

As done for strong bisimilarity, we can now define a transformation function $\Psi : \wp(\mathscr{P} \times \mathscr{P}) \to \wp(\mathscr{P} \times \mathscr{P})$ which takes a relation on $\mathscr{P}$ and returns another relation $\Psi(R)$ by exploiting simulations via weak transitions:

$$p\ \Psi(R)\ q \overset{\text{def}}{=} \begin{cases} \forall \mu, p'.\ p \xrightarrow{\mu} p' \text{ implies } \exists q'.\ q \overset{\mu}{\Rightarrow} q' \text{ and } p'\ R\ q' \\ \forall \mu, q'.\ q \xrightarrow{\mu} q' \text{ implies } \exists p'.\ p \overset{\mu}{\Rightarrow} p' \text{ and } p'\ R\ q' \end{cases}$$

Then a weak bisimulation R is just a relation such that $\Psi(R) \sqsubseteq R$ (i.e., $R \subseteq \Psi(R)$). From which it follows:

$$p \approx q \quad \text{if and only if} \quad \exists R.\ p\,R\,q \wedge \Psi(R) \sqsubseteq R$$

and that an alternative definition of weak bisimilarity is

$$p \approx q \stackrel{\text{def}}{=} \bigsqcup_{\Psi(R) \sqsubseteq R} R.$$

Weak bisimilarity seems to improve the notion of equivalence w.r.t. $\simeq$, because $\approx$ abstracts away from the invisible transitions as we required. Unfortunately, there are two problems with this relation:

1. First, the LTS obtained by considering weak transitions instead of string transitions can become infinitely branching also for guarded terms (consider, e.g., the finitely branching process **rec** $x.\ (\tau.x \mid \alpha.\mathbf{nil})$, analogous to the agent discussed in Example 11.13). Thus function Ψ is not continuous, and the minimal fixpoint cannot be reached, in general, with the usual chain of approximations.
2. Second, and much worse, weak bisimilarity is not a congruence with respect to the choice operator $+$, as the following example shows. As a (minor) consequence, weak bisimilarity, differently than strong bisimilarity, cannot be axiomatised by context-insensitive laws.

Example 11.19 (Weak bisimilarity is not a congruence). Let p and q be the following CCS agents:

$$p \stackrel{\text{def}}{=} \alpha.\mathbf{nil} \qquad q \stackrel{\text{def}}{=} \tau.\alpha.\mathbf{nil}$$

Obviously, we have $p \approx q$, since their behaviours differ only by the ability to perform an invisible action τ. Now we define the following context:

$$C[\cdot] = \cdot + \beta.\mathbf{nil}$$

Then by embedding p and q within the context $C[\cdot]$ we get:

$$C[p] = \alpha.\mathbf{nil} + \beta.\mathbf{nil} \not\approx \tau.\alpha.\mathbf{nil} + \beta.\mathbf{nil} = C[q]$$

In fact $C[q] \xrightarrow{\tau} \alpha.\mathbf{nil}$, while $C[p]$ has only one invisible weak transition that can be used to match such a step, that is the idle step $C[p] \stackrel{\tau}{\Rightarrow} C[p]$ and $C[p]$ is clearly not equivalent to $\alpha.\mathbf{nil}$ (because the former can perform a β-transition that the latter cannot simulate). This phenomenon is due to the fact that τ-transitions are not observable but can be used to discard some alternatives within non-deterministic choices. While quite unpleasant, the above fact is not in any way due to a CCS weakness, or misrepresentation of reality, but rather enlightens a general property of nondeterministic choice in systems represented as black boxes.

Note, however, that weak bisimilarity is an equivalence relation and that it is a congruence w.r.t. all operators, except sum.

11.8.2 Weak Observational Congruence

As shown by the Example 11.19, weak bisimilarity is not a congruence relation. In this section we present one possible (partial) solution. The idea is to close the equivalence w.r.t. all sum contexts.

Let us consider the Example 11.19, where the execution of a τ-transition forces the system to make a choice which is invisible to an external observer. In order to make this kind of choices observable we can define the relation $\approxeq$ as follows

Definition 11.17 (Weak observational congruence $\approxeq$). We say that two processes p and q are *weakly observational congruent*, written $p \approxeq q$ if

$$p \approx q \ \wedge \ \forall r \in \mathscr{P}.\ p + r \approx q + r.$$

Weak observational congruence can be defined directly by letting:

$$p \approxeq q \ \stackrel{\text{def}}{=} \begin{cases} \forall p'.\ p \xrightarrow{\tau} p' \ \text{ implies } \ \exists q'.\ q \xrightarrow{\tau}\overset{\tau}{\Rightarrow} q' \text{ and } p' \approx q' \\ \forall \lambda, p'.\ p \xrightarrow{\lambda} p' \ \text{ implies } \ \exists q'.\ q \overset{\lambda}{\Rightarrow} q' \text{ and } p' \approx q' \\ \text{(and, vice versa, any transition of } q \text{ can be (weakly) simulated by } p\text{)} \end{cases}$$

As we can see, an internal action $p \xrightarrow{\tau} p'$ must now be matched by at least one internal action. Notice however that this is not a recursive definition, since $\approxeq$ is simply defined in terms of $\approx$: after the first step has been performed, other τ-labeled transition can be simulated also by staying idle. Now it is obvious that $\alpha.\,\mathbf{nil} \not\approxeq \tau.\alpha.\,\mathbf{nil}$, because $\alpha.\,\mathbf{nil}$ cannot simulate the τ-transition $\tau.\alpha.\,\mathbf{nil} \xrightarrow{\tau} \alpha.\,\mathbf{nil}$.

The relation $\approxeq$ is a congruence but as we can see in the following example it is not a (weak) bisimulation, namely $\approxeq \not\subseteq \Psi(\approxeq)$.

Example 11.20 (Weak observational congruence is not a weak bisimulation). Let

$$p \stackrel{\text{def}}{=} \beta.p' \qquad p' \stackrel{\text{def}}{=} \tau.\alpha.\,\mathbf{nil} \qquad q \stackrel{\text{def}}{=} \beta.q' \qquad q' \stackrel{\text{def}}{=} \alpha.\,\mathbf{nil}$$

We have $p' \not\approxeq q'$ (see above), although Example 11.19 shows that $p' \approx q'$. Therefore:

$$p \approx q \qquad \text{and} \qquad p \approxeq q$$

but, according to the weak bisimulation game, if Alice the attacker plays the β-transition $p \xrightarrow{\beta} p'$, Bob the defender has no chance of playing a (weak) β-transition on q and reach a state that is related by $\approxeq$ with p'. Thus $\approxeq$ is not a pre-fixpoint of Ψ.

Weak observational congruence $\approxeq$ can be axiomatised by adding to the axioms for strong bisimilarity the following three Milner's τ laws:

$$p + \tau.p \approxeq \tau.p \qquad (11.1)$$

$$\mu.(p + \tau.q) \approxeq \mu.(p + \tau.q) + \mu.q \qquad (11.2)$$

$$\mu.\tau.p \approxeq \mu.p \qquad (11.3)$$

11.8.3 Dynamic Bisimilarity

Example 11.20 shows that weak observational congruence is not a (weak) bisimulation. In this section we present the largest relation which is at the same time a congruence and a weak bisimulation. It is called *dynamic bisimilarity* and was introduced by Vladimiro Sassone.

Definition 11.18 (Dynamic bisimilarity $\cong$). We define the dynamic bisimilarity $\cong$ as the largest relation that satisfies:

$$p \cong q \quad \text{implies} \quad \forall C[\cdot].\ C[p]\ \Psi(\cong)\ C[q]$$

In this case, at every step we close the relation by comparing the behaviour w.r.t. any possible embedding context. In terms of game theory this definition can be viewed as "at each turn Alice the attacker is also allowed to insert both agents into the same context and then choose the transition."

Alternatively, we can define the dynamic bisimilarity in terms of the transformation function $\Theta : \wp(\mathscr{P} \times \mathscr{P}) \to \wp(\mathscr{P} \times \mathscr{P})$ such that:

$$p\,\Theta(R)\,q \overset{\text{def}}{=} \begin{cases} \forall p'.\ p \xrightarrow{\tau} p' \ \text{ implies } \ \exists q'.\ q \xrightarrow{\tau}\overset{\tau}{\Longrightarrow} q' \ \text{ and } \ p'\,R\,q' \\ \forall \lambda, p'.\ p \xrightarrow{\lambda} p' \ \text{ implies } \ \exists q'.\ q \overset{\lambda}{\Longrightarrow} q' \ \text{ and } \ p'\,R\,q' \\ \text{(and, vice versa, any transition of } q \text{ can be (weakly) simulated by } p\text{)} \end{cases}$$

In this case, every internal move must be simulated by making at least one internal move: this is different from weak observational congruence, where after the first step, an internal move can be simulated by staying idle, and it is also different from weak bisimulation, where any internal move can be simulated by staying idle.

Then, we say that R is a *dynamic bisimulation* if $\Theta(R) \sqsubseteq R$, and *dynamic bisimilarity* can be defined by letting:

$$\cong \overset{\text{def}}{=} \bigcup_{\Theta(R) \sqsubseteq R} R$$

Example 11.21. Let p, p', q and q' be defined as in Example 11.20. We have:

$p \approx q$	and	$p' \approx q'$	(weak bisimilarity)
$p \cong q$	and	$p' \not\cong q'$	(weak observational congruence)
$p \not\cong q$	and	$p' \not\cong q'$	(dynamic bisimilarity)

As for weak observational congruence, we can axiomatise dynamic bisimilarity of finite processes. The axiomatisation of $\cong$ is obtained from that of $\approxeq$ by omitting the third Milner's τ law (Equation 11.3), i.e., by adding to the axioms for strong bisimilarity the laws:

$$p + \tau.p \cong \tau.p \tag{11.4}$$

$$\mu.(p + \tau.q) \cong \mu.(p + \tau.q) + \mu.q \tag{11.5}$$

Problems

11.1. Draw the complete LTS for the agent of Example 11.2.

11.2. Write the recursive CCS process that corresponds to X_3 in Example 11.5.

11.3. Given a natural number $n \geq 1$, let us define the family of CCS processes B^n_k for $0 \leq k \leq n$ by letting:

$$B^n_0 \overset{\text{def}}{=} in.B^n_1 \qquad B^n_k \overset{\text{def}}{=} in.B^n_{k+1} + \overline{out}.B^n_{k-1} \text{ for } 0 < k < n \qquad B^n_n \overset{\text{def}}{=} \overline{out}.B^n_{n-1}$$

Intuitively B^n_k represents a buffer with n positions of which k are occupied (see Example 11.8).

Prove that $B^n_0 \simeq \underbrace{B^1_0 \mid B^1_0 \mid \cdots \mid B^1_0}_{n}$ by providing a suitable strong bisimulation.

11.4. Prove that the union $R_1 \cup R_2$ and the composition

$$R_1 \circ R_2 \overset{\text{def}}{=} \{(p, p') \mid \exists p''.\, p\ R_1\ p'' \wedge p''\ R_2\ p'\}$$

of two strong bisimulation relations R_1 and R_2 are also strong bisimulation relations.

11.5. Exploit the properties outlined in Problem 11.4 to prove that strong bisimilarity is an equivalence relation (i.e., to prove Theorem 11.1).

11.6. CCS is expressive enough to encode language constructs from imperative programming, e.g., shared memory models of computation. A possible encoding is outlined below:

Termination: To represent sequential composition of commands, we can use a dedicated channel *done* over which a message is sent when the current command is terminated. The message will be received by the continuation. In the following we let *Done* denote the process

$$Done \overset{\text{def}}{=} \overline{done}.\mathbf{nil}$$

Variables: Suppose x is a variable whose possible values range over a finite domain $\{v_1, ..., v_n\}$. Such variables can have n different states $X_1, X_2, ..., X_n$, depending on the currently stored value. In any such state, a write operation can change the value stored in the variable, or the current value can be read. We can model this situation by considering (recursively defined processes):

$$\begin{aligned}
X_1 &\overset{\text{def}}{=} xw_1.X_1 + xw_2.X_2 + ... + xw_n.X_n + xr_1.X_1\\
X_2 &\overset{\text{def}}{=} xw_1.X_1 + xw_2.X_2 + ... + xw_n.X_n + xr_2.X_2\\
&...\\
X_n &\overset{\text{def}}{=} xw_1.X_1 + xw_2.X_2 + ... + xw_n.X_n + xr_n.X_n
\end{aligned}$$

where in any state X_i:

- for any $j \in [1,n]$ a message on channel xw_j causes a change of state to X_j;
- a message on channel xr_j is accepted if and only if $j = i$.

Allocation: A variable declaration like

$$\text{var } x$$

can be modelled by the allocation of an uninitialised variable, together with the termination message:

$$xw_1.X_1 + xw_2.X_2 + \ldots + xw_n.X_n \mid Done$$

Assignment: An assignment like

$$x := i$$

can be modelled by sending a message over the channel xw_i to the process that manages the variable x:

$$\overline{xw_i}.Done$$

Skip: A skip statement is translated directly as $\tau.Done$ or simply $Done$.

Sequencing: Let p_1, p_2 be the CCS processes modelling the commands c_1, c_2. Then, we could try to model the sequential composition

$$c_1; c_2$$

simply as $p_1 \mid done.p_2$, but this solution is unfortunate, because when considering several processes composed sequentially, like $(c_1; c_2); c_3$, then the termination signal produced by p_1 could activate p_3 instead of p_2. To amend the situation, we can introduce a channel d, private to p_1 and p_2, which is used to rename the termination channel of p_1 (while termination of p_2 is still on channel $done$):

$$(p_1[\phi_{done}] \mid d.p_2) \backslash d$$

where ϕ_{done} is the relabelling such that $\phi_{done}(done) = d$.

Complete the encoding by implementing the following constructs:

Conditionals: Let p_1, p_2 be the CCS processes modelling the commands c_1, c_2. Then, how can we model the conditional statement below?

$$\textbf{if } x = i \textbf{ then } c_1 \textbf{ else } c_2$$

Iteration: Let p be the CCS process modelling the command c. Then, how can we model the while statement below?

$$\textbf{while } x = i \textbf{ do } c$$

Concurrency: Let p_1, p_2 be the CCS processes modelling the commands c_1, c_2. Then, how can we model the parallel composition below?

$$c_1 \mid c_2$$

Hint: note that $p_1 \mid p_2$ is not the correct answer: we want to signal termination when both the executions of p_1 and p_2 are terminated.

11.7. Prove that strong bisimilarity is a congruence w.r.t. action prefix, restriction, relabelling and sum (see Section 11.5.1).

11.8. Let us consider the agent $A \stackrel{\text{def}}{=} \textbf{rec}\ x.\ (\alpha.x \mid \beta.\textbf{nil})$. Prove that among the reachable states from A there exist infinitely many states that are not strong bisimilar. Exploit this fact to conclude that A is not strong bisimilar to any agent with a finite number of reachable states.

11.9. Let us consider the CCS processes

$$p \stackrel{\text{def}}{=} \textbf{rec}\ x.\ (a.x + a.\textbf{nil}) \qquad q \stackrel{\text{def}}{=} \textbf{rec}\ y.\ (a.a.y + a.\textbf{nil})$$

Draw the LTS for p and q and prove that $p \not\simeq q$ by exhibiting a formula in HM-logic that distinguishes between the two.

11.10. Let us consider the CCS processes

$$r \stackrel{\text{def}}{=} a.(b.c.\textbf{nil} + b.\tau.c.\textbf{nil} + \tau.b.\textbf{nil} + b.\textbf{nil})$$
$$s \stackrel{\text{def}}{=} a.(b.c.\tau.\textbf{nil} + \tau.b.\textbf{nil}) + a.b.\textbf{nil}$$

Draw the LTS for r and s and prove that they are weakly observational congruent by exploiting the axioms presented in Sections 11.7 and 11.8.2. At each step of the proof explain which axiom is used and where it is applied.

11.11. Consider the CCS agents:

$$p \stackrel{\text{def}}{=} (\textbf{rec}\ x.\ a.x) \mid \textbf{rec}\ y.\ b.y \qquad q \stackrel{\text{def}}{=} \textbf{rec}\ z.\ a.a.z + a.b.z + b.a.z + b.b.z$$

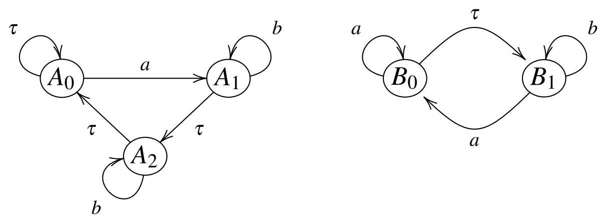
Prove that p and q are strong bisimilar or exhibit an HM-logic formula F that can be used to distinguish them.

11.12. Let us consider sequential CCS agents composed using only **nil**, action prefix and sum. Prove that

$$p \xrightarrow{\mu} q \quad \text{implies} \quad \varphi(p) \xrightarrow{\varphi(\mu)} \varphi(q)$$

for any permutation of action names φ. Use this result to prove that $p \simeq q$ implies $\varphi(p) \simeq \varphi(q)$, where $\simeq$ denotes strong bisimilarity.

11.13. Let us consider the LTSs below:



1. Write the recursive CCS expressions that corresponds to A_0 and B_0.
   Hint: Introduce a **rec** construct for each node in the diagram and name the process variables as the nodes for simplicity, e.g., for A_0 write **rec** $A_0.\ (\tau.A_0 + \ldots)$.
2. Prove that $A_0 \not\approx B_0$ and $B_0 \approx B_1$, where $\approx$ is the weak bisimilarity.

11.14. Recall that a weak bisimulation is a relation R such that:

$$\forall p,q. \quad p\,R\,q \text{ implies } \begin{cases} \forall \mu, p'. \quad p \xrightarrow{\mu} p' \quad \text{implies} \quad \exists q'.\, q \overset{\mu}{\Rightarrow} q' \quad \text{and} \quad p'\,R\,q' \\ \forall \mu, q'. \quad q \xrightarrow{\mu} q' \quad \text{implies} \quad \exists p'.\, p \overset{\mu}{\Rightarrow} p' \quad \text{and} \quad p'\,R\,q' \end{cases}$$

Let us define a *loose bisimulation* to be a relation R such that:

$$\forall p,q. \quad p\,R\,q \text{ implies } \begin{cases} \forall \mu, p'. \quad p \overset{\mu}{\Rightarrow} p' \quad \text{implies} \quad \exists q'.\, q \overset{\mu}{\Rightarrow} q' \quad \text{and} \quad p'\,R\,q' \\ \forall \mu, q'. \quad q \overset{\mu}{\Rightarrow} q' \quad \text{implies} \quad \exists p'.\, p \overset{\mu}{\Rightarrow} p' \quad \text{and} \quad p'\,R\,q' \end{cases}$$

Prove that weak bisimilarity is the largest loose bisimulation by showing that:

1. any loose bisimulation is a weak bisimulation; and
2. any weak bisimulation is a loose bisimulation.

Hint: For (2) prove first, by mathematical induction on $n \geq 0$, that for any weak bisimulation R, any two processes $p\,R\,q$, and any sequence of transitions $p \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \cdots \xrightarrow{\tau} p_n$ there exists q' with $q \overset{\tau}{\Rightarrow} q'$ and $p_n\,R\,q'$.

11.15. Let $\mathscr{P}$ denote the set of all (closed) CCS processes.

1. Prove that $\forall p,q \in \mathscr{P}.\ p \mid q \approx q \mid \tau.p$, where $\approx$ denotes weak bisimilarity, by showing that the relation R below is a weak bisimulation:

$$R \overset{\text{def}}{=} \{(p \mid q, q \mid \tau.p) \mid p,q \in \mathscr{P}\} \cup \{(p \mid q, q \mid p) \mid p,q \in \mathscr{P}\}$$

2. Then exhibit two processes p and q and a context $C[\cdot]$ showing that $s \overset{\text{def}}{=} p \mid q$ and $t \overset{\text{def}}{=} q \mid \tau.p$ are not weak observational congruent.
   Hint: remind that, denoting by $\approxeq$ the weak observational congruence:

$$s \approxeq t \text{ if and only if } s \approx t \wedge \forall r.\, s + r \approx t + r.$$

11.16. Prove that for any HM-formula F we have $(F^c)^c = F$ and $\mathrm{md}(F^c) = \mathrm{md}(F)$.