Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

May 26, 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.

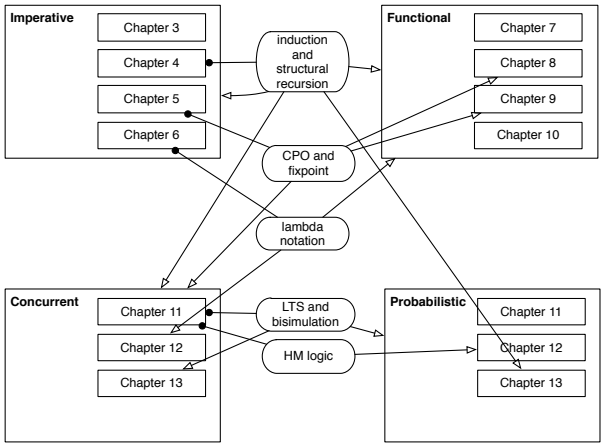
Each of the above models can be studied in separation from the others, but previous parts introduce a body of notions and techniques that are also applied and extended in later parts.

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)
$\cong$	CCS weak observational congruence (see Section 11.8.2)
$\mathrel{\hat{=}}$	CCS dynamic bisimilarity (see Definition 11.18)
$\sim_E$	π-calculus strong early bisimilarity (see Definition 13.3)
$\sim_L$	π-calculus strong late bisimilarity (see Definition 13.4)
$\simeq_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\simeq_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\approx_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\approx_L$	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)
$\text{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.2.2)
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 12.2.1)
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 14.4.2)
$\mathbb{R}$	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 of CCS, 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 necessarily 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. In particular we study bisimilarity, a milestone abstract equivalence with large applicability and interesting theoretical properties. We also define a suitable modal logic, called Hennessy-Milner logic, whose induced logical equivalence is shown to coincide with 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 (e.g., the IMP command **while true do skip** and the HOFL term **rec** $x.\, x$) is assigned the denotation $\bot$, 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.

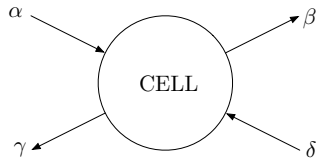
Let us show how CCS works with an example.

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.

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

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 $\text{CELL}_1(x)$.

[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.

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

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 enters the state $\mathrm{CELL}_2(y,v)$. 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.

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

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 $\mathrm{CELL}_1(u)$.

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

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**.

Notice that ENDCELL cannot send or receive messages on its β and δ channels. In fact, ENDCELL should possess no such channels. Also, the behaviour of the stack is correct only if the initial state of the agent is ENDCELL.

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).

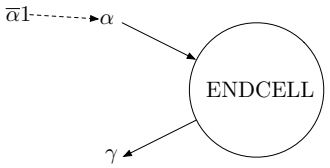


Fig. 11.1: ENDCELL$\backslash\beta\backslash\delta$ receiving the value 1 on channel α

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.

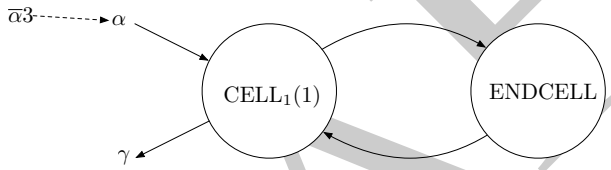


Fig. 11.2: $(\text{CELL}_1(1) \supset \text{ENDCELL})\backslash\beta\backslash\delta$ receiving the value 3 on channel α

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).

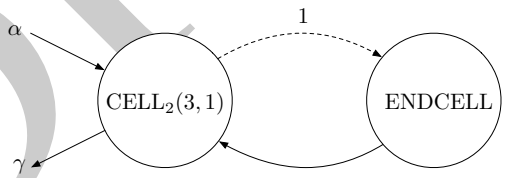


Fig. 11.3: $(\text{CELL}_2(3,1) \supset \text{ENDCELL})\backslash\beta\backslash\delta$ before right-shifting the value 1

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 $\supset$, it will change its state to $\text{CELL}_1(1)$ and will spawn a new

ENDCELL, while the leftmost cell moves from the state $\mathrm{CELL}_2(3,1)$ to the state $\mathrm{CELL}_1(3)$ (see Figure 11.4). Note that the linking operation is associative.

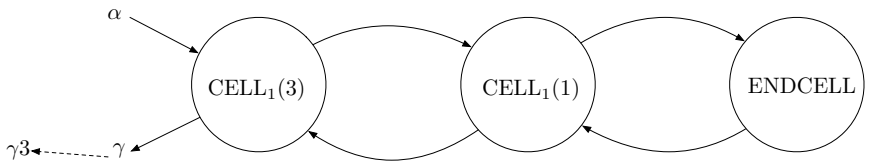


Fig. 11.4: $\mathrm{CELL}_1(3) \supset \mathrm{CELL}_1(1) \supset \mathrm{ENDCELL} \backslash \beta \backslash \delta$ before a pop operation

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 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.

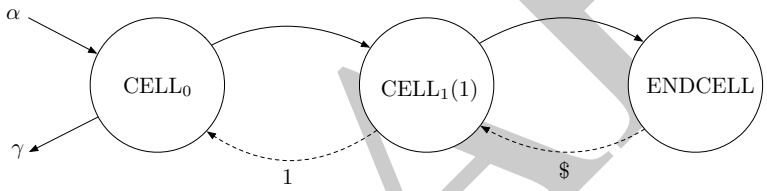


Fig. 11.5: $(\mathrm{CELL}_0 \supset \mathrm{CELL}_1(1) \supset \mathrm{ENDCELL}) \backslash \beta \backslash \delta$ before left-shifting value 1

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, because it is the unit of composition).

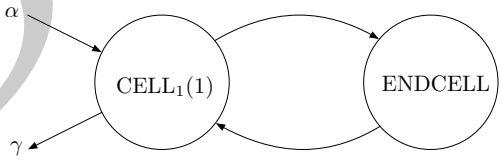


Fig. 11.6: $(\mathrm{CELL}_1(1) \supset \mathrm{ENDCELL} \supset \mathbf{nil}) \backslash \beta \backslash \delta$

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:

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

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 $p\backslash\alpha$ allows synchronisations on α that are internal to p, but disallows 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 executed, 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 $p \stackrel{\text{def}}{=} ((\alpha.\mathbf{nil} + \overline{\beta}.\mathbf{nil})[\phi] + \alpha.\mathbf{nil})\backslash\alpha$ 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.

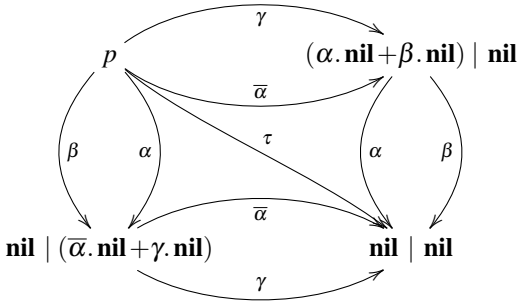
$$\text{(Par)} \quad \frac{p \xrightarrow{\mu} p'}{p \mid q \xrightarrow{\mu} p' \mid q} \qquad \frac{q \xrightarrow{\mu} q'}{p \mid q \xrightarrow{\mu} p \mid q'}$$

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.

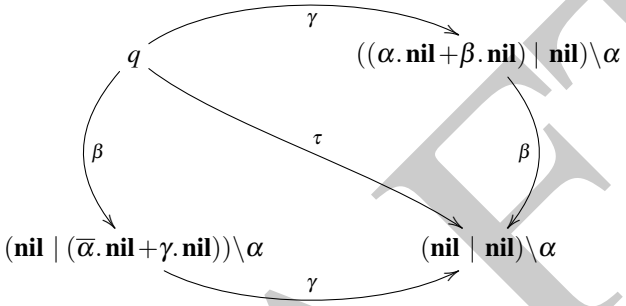
$$\text{(Com)} \quad \frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\overline{\lambda}} q_2}{p_1 \mid q_1 \xrightarrow{\tau} p_2 \mid q_2}$$

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 $p \stackrel{\text{def}}{=} (\alpha.\mathbf{nil} + \beta.\mathbf{nil}) \mid (\overline{\alpha}.\mathbf{nil} + \gamma.\mathbf{nil})$ is:



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



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 $(\alpha.x)[^{\mathbf{rec}\ x.\ \alpha.x}/_x] = \alpha.\mathbf{rec}\ x.\ \alpha.x$. Namely, 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

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

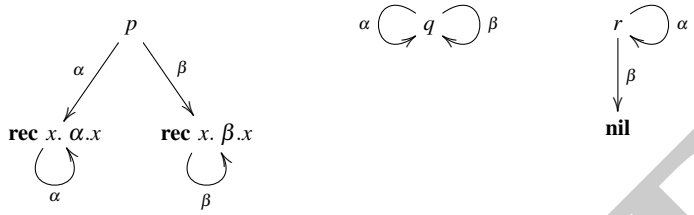


Fig. 11.7: The LTSs of three recursively defined processes

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 never terminate, while r may or may not terminate.

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 $\mathbf{rec}\ x.\ p$, 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 $\mathbf{rec}\ x.\ (x \mid p)$ 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, we define the predicate $G(p,X)$ for any process p and set of process names X as follows:

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

The predicate $G(p,X)$ is true if and only if (i) every process name in X is either not free in p or free and prefixed by an action; and (ii) all recursively defined names in p occur guarded in p.

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, $G(p,X) \Rightarrow G(p,X \cup \{x\})$ if 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:

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

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

$$\mathbf{rec}\ x.\ \overline{\alpha}.x \xrightarrow{\lambda} r' \qquad \nwarrow_{\text{Rec}}\ \overline{\alpha}.(\mathbf{rec}\ x.\ \overline{\alpha}.x) \xrightarrow{\lambda} r'$$
$$\nwarrow_{\text{Act},\ \lambda=\overline{\alpha},\ r'=\mathbf{rec}\ x.\ \overline{\alpha}.x}\ \square$$

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.:

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

From which we derive:

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

and thus:

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

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 switches β with ϑ, δ with η and is the identity otherwise, while $\phi_{\alpha,\gamma}$ switches α with ϑ, γ with 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).

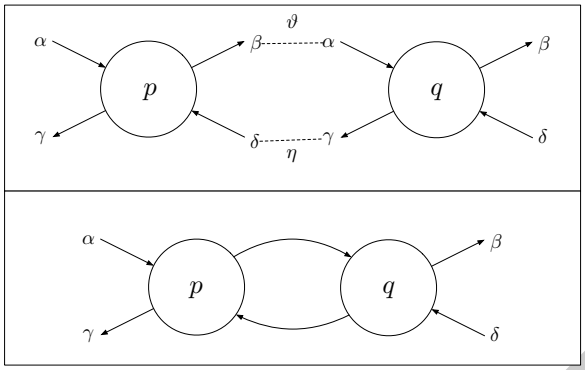


Fig. 11.8: Graphically illustration of the concatenation operator $p \supset q$

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

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

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:

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

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

$$q_i \stackrel{\text{def}}{=} \mathbf{rec}\ X_i.\ p_i$$

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

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

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 $X_2, ..., X_n$.

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

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

Then we have

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

From which we derive

$$r_3 \stackrel{\text{def}}{=} q_3 = \textbf{rec}\ X_3.\ \delta.X_2$$
$$r_2 \stackrel{\text{def}}{=} q_2[r_3/X_3] = \textbf{rec}\ X_2.\ (\beta.X_1 + \gamma.\textbf{rec}\ X_3.\ \delta.X_2)$$
$$r_1 \stackrel{\text{def}}{=} q_1[r_3/X_3][r_2/X_2] = \textbf{rec}\ X_1.\ \alpha.\textbf{rec}\ X_2.\ (\beta.X_1 + \gamma.\textbf{rec}\ X_3.\ \delta.X_2)$$

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 $p+\textbf{nil}$ 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.

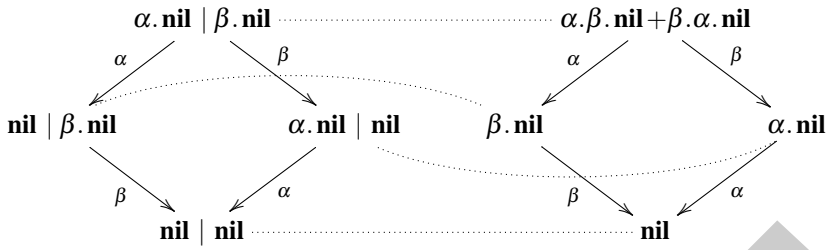
In this section, we first show that neither graph isomorphism nor trace equivalence offer fully satisfactory abstract semantics for CCS. Next, we introduce a more appropriate abstract semantics of CCS by defining a relation, called *strong bisimilarity*, that captures the ability of processes to simulate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

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*. We discuss compositionality issues in Section 11.5.

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 $\alpha.\textbf{nil} \mid \beta.\textbf{nil}$ and $\alpha.\beta.\textbf{nil} + \beta.\alpha.\textbf{nil}$. Their LTSs are as follows:



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 **rec** $x.\ \alpha.x$, **rec** $x.\ \alpha.\alpha.x$ and $\alpha.$**rec** $x.\ \alpha.x$, whose LTSs are in Figure 11.9:

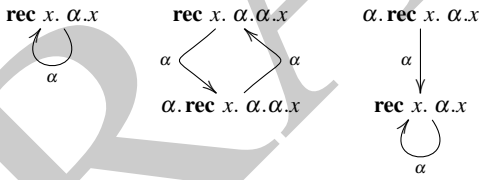


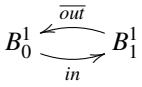
Fig. 11.9: Three non-isomorphic agents

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):

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

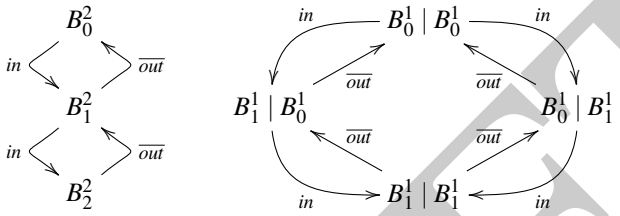
The corresponding LTS is



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

$$B_0^2 \stackrel{\text{def}}{=} in.B_1^2 \qquad B_1^2 \stackrel{\text{def}}{=} \overline{out}.B_0^2 + in.B_2^2 \qquad B_2^2 \stackrel{\text{def}}{=} \overline{out}.B_1^2$$

Another possibility for obtaining an (empty) buffer of capacity 2 is to use two (empty) buffers of capacity 1 composed in parallel: $B_0^1 \mid B_0^1$. However the LTSs of B_0^2 and $B_0^1 \mid B_0^1$ are not isomorphic, because they have a different number of states:



The LTS of B_0^2 offers a minimal realisation of the behaviour of the buffer: the three states B_0^2, B_1^2 and B_2^2 cannot be identified, because they exhibit different behaviours (e.g., B_2^2 cannot perform an *in* action, unlike B_1^2 and B_0^2, while B_0^2 can perform two *in* actions in a row, unlike B_1^2 and B_2^2). Instead, the LTS of $B_0^1 \mid B_0^1$ has two different states that should be considered as equivalent, namely $B_1^1 \mid B_0^1$ and $B_0^1 \mid B_1^1$ (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 $p_1, ..., p_k$. 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.

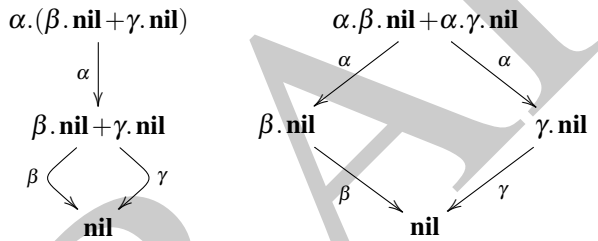
Definition 11.3 (Context). A *context* is a term with a hole which can be filled by inserting any other term of our language.

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:

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

Their LTSs are as follows:



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., an agent

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

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

$$(p \mid r)\backslash\alpha\backslash\beta\backslash\gamma \qquad (q \mid r)\backslash\alpha\backslash\beta\backslash\gamma.$$

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

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

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 intended 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 $p \stackrel{\alpha}{\to} \beta.\mathbf{nil}+\gamma.\mathbf{nil}$. Then, Bob can choose one of the two α-transitions leaving from q. Suppose that Bob chooses the α-transition $q \stackrel{\alpha}{\to} \beta.\mathbf{nil}$ (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 $\beta.\mathbf{nil}+\gamma.\mathbf{nil} \stackrel{\gamma}{\to} \mathbf{nil}$, 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

$$\forall s_1, s_2.\ s_1\ R\ s_2 \Rightarrow \begin{cases} \forall \mu, s_1'.\ s_1 \stackrel{\mu}{\to} s_1' \ \text{ implies } \ \exists s_2'.\ s_2 \stackrel{\mu}{\to} s_2' \text{ and } s_1'\ R\ s_2'; \text{ and} \\ \forall \mu, s_2'.\ s_2 \stackrel{\mu}{\to} s_2' \ \text{ implies } \ \exists s_1'.\ s_1 \stackrel{\mu}{\to} s_1' \text{ and } s_1'\ R\ s_2'. \end{cases}$$

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:

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

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: $\forall x \in X.\ x \equiv x$;
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:*

$$s_1 \simeq s_2 \quad \Leftrightarrow \quad \begin{cases} \forall \mu, s_1'.\ s_1 \xrightarrow{\mu} s_1' \ \textit{ implies } \ \exists s_2'.\ s_2 \xrightarrow{\mu} s_2' \textit{ and } s_1' \simeq s_2'; \ \textit{ and} \\ \forall \mu, s_2'.\ s_2 \xrightarrow{\mu} s_2' \ \textit{ implies } \ \exists s_1'.\ s_1 \xrightarrow{\mu} s_1' \textit{ and } s_1' \simeq s_2'. \end{cases}$$

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 for strong bisimulation. It is immediate 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 powerset $\wp(\mathscr{P} \times \mathscr{P})$ of pairs of CCS processes, i.e., the set of all relations on $\mathscr{P}$. 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 $CPO_{\perp}$ $(\wp(\mathscr{P} \times \mathscr{P}), \sqsubseteq)$ is represented in Figure 11.10.

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

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

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. □

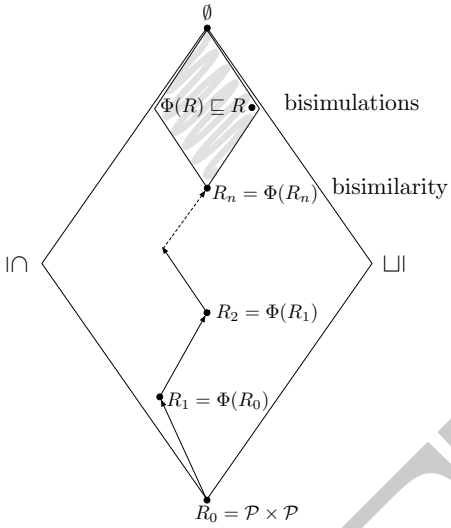


Fig. 11.10: The $CPO_\perp$ $(\wp(\mathscr{P} \times \mathscr{P}), \sqsubseteq)$

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

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

Theorem 11.4. *Strong bisimilarity is 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 $s_1 \stackrel{\mu}{\to} s_1'$. Since $s_1 \; \Phi(R_2) \; s_2$, there exists s_2' such that $s_2 \stackrel{\mu}{\to} s_2'$ 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 $s_2 \stackrel{\mu}{\to} s_2'$. □

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 $r \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ \tau.x$; it can only execute infinitely many τ-transitions. Now, for $n \in \mathbb{N}$, let $p_n \stackrel{\text{def}}{=} \underbrace{\tau....\tau}_{n \text{ times}}.\mathbf{nil}$ 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,

$$p = p_1 + p_2 + ... + p_n + ...$$

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 that switches α with β and take $p = (p' \mid \alpha.\mathbf{nil})\backslash\alpha$, where:

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

The process p' can execute any sequence of τ-transitions concluded by an $\overline{\alpha}$-transition, but when performing the first transition it is left with the possibility to execute as many transitions as the number of times the recursive definition has been unfolded. To see this, observe that clearly $p' \xrightarrow{\overline{\alpha}} \mathbf{nil}$. Therefore

$$\begin{array}{rl}
p' \xrightarrow{\lambda} s & \nwarrow^{*}_{Rec,\ Sum}\ (p'[\phi] \mid \beta.\overline{\alpha}.\mathbf{nil})\backslash\beta \xrightarrow{\lambda} s \\
& \nwarrow_{Res,\ s=s_1\backslash\beta}\ p'[\phi] \mid \beta.\overline{\alpha}.\mathbf{nil} \xrightarrow{\lambda} s_1, \quad \lambda \neq \beta, \overline{\beta} \\
& \nwarrow_{Com,\ \lambda=\tau,\ s_1=s_2|s_3}\ p'[\phi] \xrightarrow{\lambda_1} s_2, \quad \beta.\overline{\alpha}.\mathbf{nil} \xrightarrow{\overline{\lambda_1}} s_3 \\
& \nwarrow_{Rel,\ \lambda_1=\phi(\lambda_2),\ s_2=s_4[\phi]}\ p' \xrightarrow{\lambda_2} s_4, \quad \beta.\overline{\alpha}.\mathbf{nil} \xrightarrow{\overline{\phi(\lambda_2)}} s_3 \\
& \nwarrow^{*}_{\lambda_2=\overline{\alpha},\ s_4=\mathbf{nil}}\ \beta.\overline{\alpha}.\mathbf{nil} \xrightarrow{\beta} s_3 \\
& \nwarrow_{Act,\ s_3=\overline{\alpha}.\mathbf{nil}}\ \square
\end{array}$$

That is $p' \xrightarrow{\tau} (\mathbf{nil}[\phi]|\overline{\alpha}.\mathbf{nil})\backslash\beta \xrightarrow{\overline{\alpha}} (\mathbf{nil}[\phi]|\mathbf{nil})\backslash\beta$. Then, we have

$$\begin{array}{rl}
p' \xrightarrow{\lambda} s & \nwarrow^{*}_{Rec,\ Sum,\ Res,\ s=s_1\backslash\beta}\ p'[\phi] \mid \beta.\overline{\alpha}.\mathbf{nil} \xrightarrow{\lambda} s_1, \quad \lambda \neq \beta, \overline{\beta} \\
& \nwarrow_{Par,\ s_1=s_2|\beta.\overline{\alpha}.\mathbf{nil}}\ p'[\phi] \xrightarrow{\lambda} s_2, \quad \lambda \neq \beta, \overline{\beta} \\
& \nwarrow_{Rel,\ \lambda=\phi(\lambda_1)\ s_2=s_3[\phi]}\ p' \xrightarrow{\lambda_1} s_3, \quad \phi(\lambda_1) \neq \beta, \overline{\beta} \\
& \nwarrow^{*}_{\lambda_1=\tau,\ s_3=(\mathbf{nil}[\phi]|\overline{\alpha}.\mathbf{nil})\backslash\beta}\ \square
\end{array}$$

So $p' \xrightarrow{\tau} (((\mathbf{nil}[\phi]|\overline{\alpha}.\mathbf{nil})\backslash\beta)[\phi]|\beta.\overline{\alpha}.\mathbf{nil})\backslash\beta \xrightarrow{\tau} (((\mathbf{nil}[\phi]|\mathbf{nil})\backslash\beta)[\phi]|\overline{\alpha}.\mathbf{nil})\backslash\beta \xrightarrow{\overline{\alpha}} (((\mathbf{nil}[\phi]|\mathbf{nil})\backslash\beta)[\phi]|\mathbf{nil})\backslash\beta$, and so on.

Now, for any $n \in \mathbb{N}$, let $\simeq_n \stackrel{\text{def}}{=} \Phi^n(\mathscr{P} \times \mathscr{P})$. 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 $\simeq_0 \stackrel{\text{def}}{=} \mathscr{P} \times \mathscr{P}$. 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., $p \simeq_{n+1} r$.

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 and*

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

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:

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

Note that, in the $\mathrm{CPO}_\perp$ $(\wp(\mathscr{P}_f \times \mathscr{P}_f), \sqsubseteq)$, the least upper bound $\bigsqcup_{n\in\mathbb{N}} R_n$ of a chain of relations is obtained by taking the intersection of all relations in the chain, not their union. 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. □

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 $X = \{x_1,...,x_n\}$ of process names and processes $p_1,...,p_n$, then $G(p,X) \wedge \bigwedge_{i\in[1,n]} G(p_i,X)$ implies that the set $\{q \mid p[{}^{p_1}/_{x_1},...,{}^{p_n}/_{x_n}] \xrightarrow{\mu} q\}$ is finite. The proof is by structural induction on p. For brevity, let σ denote the substitution $[{}^{p_1}/_{x_1},...,{}^{p_n}/_{x_n}]$. 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 $p = \mathbf{rec}\ x.\ p'$ such that $G(p',X\cup\{x\})$,[3] then the set

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

is finite by inductive hypothesis. □

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

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

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

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.:

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

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

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

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 $p_1,\ldots,p_n$ running in parallel, each with k possibly reachable states, then the process $((p_1 \mid p_2) \mid \ldots p_n)$ can have up to k^n reachable states.

Example 11.14 (Strong bisimilarity as least 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:

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

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.

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

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, $\beta.\mathbf{nil}$ and $\gamma.\mathbf{nil}$ must 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 $\mathbf{nil}$ 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:*

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

[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.

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

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

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 with respect to all CCS contexts.

The next example shows an equivalence relation that 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:

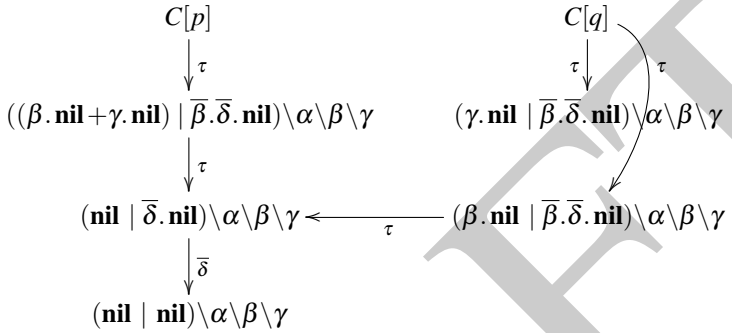
$$C[\cdot] \stackrel{\text{def}}{=} (\ \cdot \mid \overline{\alpha}.\overline{\beta}.\overline{\delta}.\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

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:



The difference can be formalised if we consider *completed trace semantics*. Let us write $p \not\rightarrow$ for the predicate $\neg(\exists \mu, q.\ p \xrightarrow{\mu} q)$. A *completed trace* of p is a sequence of actions $\mu_1 \cdots \mu_k$ (for $k \geq 0$) such that there exist $p_0, ..., p_k$ with

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

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 $s = s'_0 \mid s'_1$. □

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 (HM-logic). 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 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.

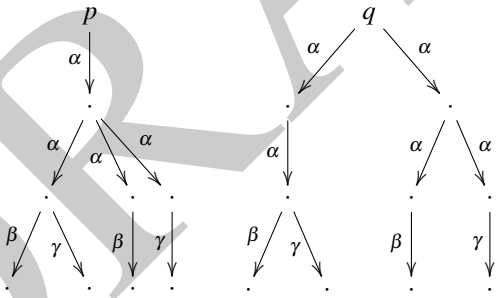
Notably, if p cannot execute any μ-transition, then $p \models \Box_\mu F$ for any formula F. For example, the formula $\Diamond_\alpha \mathit{true}$ is satisfied by all processes that can execute an α-transition, and the formula $\Box_\beta \mathit{false}$ is satisfied by all processes that cannot 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:

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

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:

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

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 12.3, 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 $F^{\mathrm{c}}_{q'}$.

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 cases, 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. □

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. Finally, the axioms that characterise strong bisimilarity 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. □

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

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

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 processes 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).

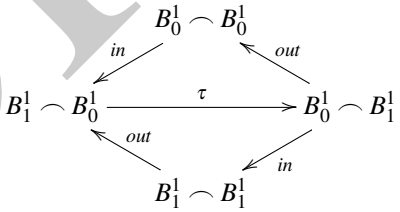
11.8 Weak Semantics of CCS

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 ϕ_{out} is the permutation that switches *out* with ℓ and ϕ_{in} is the permutation that switches *in* with ℓ (they are the identity otherwise). Then, an empty buffer of capacity two could be implemented by taking $B_0^1 \frown B_0^1$. However, its LTS is



Obviously the internal τ-transition $B_1^1 \frown B_0^1 \stackrel{\tau}{\to} B_0^1 \frown B_1^1$, 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_0^2 and $B_0^1 \frown B_0^1$.

The above example shows that, when we consider τ as an internal action, not visible from outside of the system, we would like, accordingly, to relate observable behaviours that differ 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 provably "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:

$$p \stackrel{\tau}{\Rightarrow} q \stackrel{\text{def}}{=} p \stackrel{\tau}{\rightarrow} \ldots \stackrel{\tau}{\rightarrow} q \vee p = q$$
$$p \stackrel{\lambda}{\Rightarrow} q \stackrel{\text{def}}{=} \exists p', q'.\ p \stackrel{\tau}{\Rightarrow} p' \stackrel{\lambda}{\rightarrow} q' \stackrel{\tau}{\Rightarrow} q$$

Note that $p \stackrel{\tau}{\Rightarrow} q$ means that q can be reached from p via a, possibly empty, finite sequence of τ-transitions, i.e., the weak transition relation $\stackrel{\tau}{\Rightarrow}$ coincides with the reflexive and transitive closure $(\stackrel{\tau}{\rightarrow})^*$ of the silent transition relation $\stackrel{\tau}{\rightarrow}$. For λ an observable action, the relation $\stackrel{\lambda}{\Rightarrow}$ 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

$$\forall s_1, s_2.\ s_1\ R\ s_2 \Rightarrow \begin{cases} \forall \mu, s_1'.\ s_1 \stackrel{\mu}{\rightarrow} s_1' \ \text{ implies } \ \exists s_2'.\ s_2 \stackrel{\mu}{\Rightarrow} s_2' \text{ and } s_1'\ R\ s_2'; \ \text{ and} \\ \forall \mu, s_2'.\ s_2 \stackrel{\mu}{\rightarrow} s_2' \ \text{ implies } \ \exists s_1'.\ s_1 \stackrel{\mu}{\Rightarrow} s_1' \text{ and } s_1'\ R\ s_2'. \end{cases}$$

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 $\Psi : \wp(\mathscr{P} \times \mathscr{P}) \rightarrow \wp(\mathscr{P} \times \mathscr{P})$ which takes a relation R on $\mathscr{P}$ and returns another relation $\Psi(R)$ by exploiting simulations via weak transitions:

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

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

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

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

1. First, the LTS obtained by considering weak transitions $\stackrel{\mu}{\Rightarrow}$ instead of ordinary transitions $\xrightarrow{\mu}$ 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 a congruence w.r.t. all operators, except choice $+$, as the following example shows. As a (minor) consequence, weak bisimilarity cannot be axiomatised by context-insensitive laws.

Example 11.19 (Weak bisimilarity is not a congruence). Take the 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.

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 $\approx\!\!\!\!\approx$ as follows

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

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

Weak observational congruence can be defined directly by letting:

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

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 $\approx\!\!\!\!\approx$ 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 $\alpha.\mathbf{nil} \not\approx\!\!\!\!\approx \tau.\alpha.\mathbf{nil}$, because $\alpha.\mathbf{nil}$ cannot simulate the τ-transition $\tau.\alpha.\mathbf{nil} \xrightarrow{\tau} \alpha.\mathbf{nil}$.

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

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

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

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 $\approx\!\!\!\!\approx$ with p'. Thus $\approx\!\!\!\!\approx$ is not a pre-fixpoint of Ψ.

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

$$p + \tau.p \approx\!\!\!\!\approx \tau.p \tag{11.1}$$

$$\mu.(p + \tau.q) \approx\!\!\!\!\approx \mu.(p + \tau.q) + \mu.q \tag{11.2}$$

$$\mu.\tau.p \approx\!\!\!\!\approx \mu.p \tag{11.3}$$

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:

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

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:

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

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	$p' \ncong q'$	(weak observational congruence)
$p \ncong q$	and	$p' \ncong q'$	(dynamic bisimilarity)

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

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

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

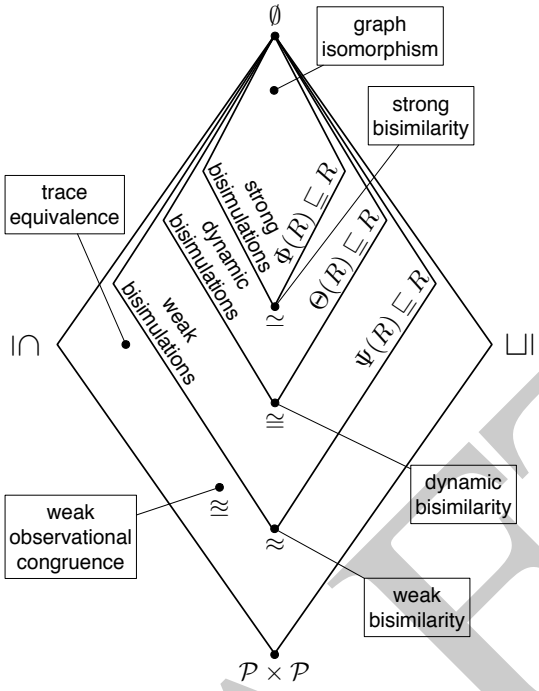


Fig. 11.11: Main relations in the $CPO_\perp$ $(\wp(\mathscr{P} \times \mathscr{P}), \sqsubseteq)$

The diagram in Figure 11.11 illustrates the main classes of bisimulations (strong, weak, and dynamic), the corresponding bisimilarities ($\simeq$, $\cong$, and $\approx$, respectively) and other notions of process equivalence (graph isomorphism, trace equivalence, and weak observational congruence). From the diagram it is evident that:

1. graph isomorphism is a strong bisimulation;
2. any strong bisimulation is also a dynamic bisimulation;
3. any dynamic bisimulation is also a weak bisimulation; and
4. all classes of bisimulations include the identity relation and are closed w.r.t. (countable) union, inverse and composition.

To memorise the various inclusions, one can note that moving from strong to dynamic bisimulation and from dynamic to weak, corresponds to allowing more options to the defender in the bisimulation game:

1. in strong games, the defender must reply by playing a single transition;
2. in dynamic games, the defender can additionally use any number of τ-transitions before and after the chosen transition;
3. in weak games, the defender can also decide to leave the process idle.

Vice versa, in all games, the rules for the attacker stay the same.

We remind that, in general, a bisimulation relation R is not an equivalence relation. However, its induced equivalence $\equiv_R$ is also a bisimulation. Moreover, all bisimilarities (i.e., the largest bisimulations) are equivalence relations. Weak bisimilarity $\approx$ is not a congruence, as marked by the absence of a bottom horizontal line in the symbol. Dynamic bisimilarity $\cong$ is the largest congruence that is also a weak bisimulation. Weak observational congruence $\cong$ is the largest congruence included in weak bisimilarity; it includes dynamic bisimilarity, but it is not a weak bisimulation. Finally, trace equivalence is a congruence relation and it includes strong bisimilarity.

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_k^n for $0 \leq k \leq n$ by letting:

$$B_0^n \stackrel{\text{def}}{=} in.B_1^n \qquad B_k^n \stackrel{\text{def}}{=} in.B_{k+1}^n + \overline{out}.B_{k-1}^n \text{ for } 0 < k < n \qquad B_n^n \stackrel{\text{def}}{=} \overline{out}.B_{n-1}^n$$

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

Prove that $B_0^n \simeq \underbrace{B_0^1 \mid B_0^1 \mid \cdots \mid B_0^1}_{n}$ by providing a suitable strong bisimulation.

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

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

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 imperative programming languages and shared memory models of computation. A possible encoding is outlined below:

Termination: We can use a dedicated channel *done* over which a message is sent when the current command is terminated. The message will trigger the continuation, if any. In the following we let:

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

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

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):

$$XW \stackrel{\text{def}}{=} \sum_{i=1}^{n} xw_i.X_i$$
$$X_1 \stackrel{\text{def}}{=} xr_1.X_1 + XW \quad ... \quad X_n \stackrel{\text{def}}{=} xr_n.X_n + XW$$

where in any state X_i and 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,[6] together with the termination message:

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

Assignment: An assignment like

$$x := v_i$$

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

$$\overline{xw_i}.Done$$

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 rename the termination channel of p_1 to a private name d, shared by p_1 and p_2 only:

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

where ϕ_{done} switches *done* with d (and is the identity otherwise).

[6] Notice that an uninitialised variable cannot be read.

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?

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

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

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

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 $\simeq$ 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. Can there exist an agent $B \simeq A$ that has a finite number of reachable states?

11.9. Prove that the LTS of any CCS process p built using only action prefix, sum, recursion and **nil** has a finite number of states.

11.10. Draw the LTS for the CCS processes

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

Then prove that $p \not\simeq q$ by exhibiting a formula in HM-logic.

11.11. Let us consider the CCS processes

$$r \stackrel{\text{def}}{=} \alpha.(\beta.\gamma.\textbf{nil} + \beta.\tau.\gamma.\textbf{nil} + \tau.\beta.\textbf{nil} + \beta.\textbf{nil}) \quad s \stackrel{\text{def}}{=} \alpha.(\beta.\gamma.\tau.\textbf{nil} + \tau.\beta.\textbf{nil}) + \alpha.\beta.\textbf{nil}$$

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.

11.12. Consider the CCS agents:

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

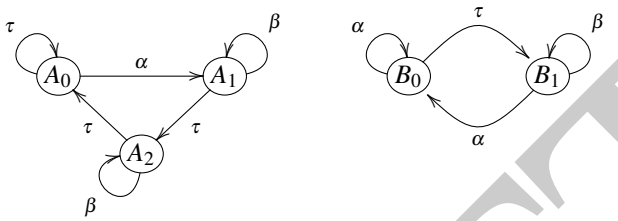
Prove that p and q are strong bisimilar or exhibit an HM-logic formula F that can be used to distinguish them.

11.13. Let us consider sequential CCS agents composed using only **nil**, action prefix and sum. Prove that for any such agents p, q and any permutation of action names φ:

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

Then prove that $p \simeq q$ implies $\varphi(p) \simeq \varphi(q)$, where $\simeq$ denotes strong bisimilarity.

11.14. 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. $(\tau.A_0 + ...)$.
2. Prove that $A_0 \not\approx B_0$ and $B_0 \approx B_1$, where $\approx$ is the weak bisimilarity.

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

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

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 $q \stackrel{\tau}{\Rightarrow} q'$ and $p_n\,R\,q'$.

11.16. 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:

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

2. Then exhibit two processes p and q and a context $C[\cdot]$ showing that $s \stackrel{\text{def}}{=} p \mid q$ and $t \stackrel{\text{def}}{=} q \mid \tau.p$ are not weak observational congruent.

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

Chapter 12
Temporal Logic and the μ-Calculus

Formal methods will never have a significant impact until they can be used by people that don't understand them. (Tom Melham)

Abstract As we have briefly discussed in the previous chapter, modal logic is a powerful tool that allows to check important behavioural properties of systems. In Section 11.6 the focus was on Hennessy-Milner logic, whose main limitation is due to its finitary structure: a formula can express properties of states up to a finite number of steps ahead and thus only local properties can be investigated. In this chapter we show some extensions of Hennessy-Milner logic that increase the expressiveness of the formulas by defining properties about finite and infinite computations. The most expressive language that we present is the *μ-calculus*, but we start by introducing some other well-known logics for program verification, called *temporal logics*.

12.1 Specification and Verification

Reactive systems, such as those composed by parallel and distributed processes, are characterised by non-terminating and highly nondeterministic behaviour. Reactive systems have become widespread in our daily activities, from banking to healthcare, and in software-controlled safety critical systems, from railways control systems to space craft control systems. Consequently, gaining maximum confidence about their trustworthiness has become an essential, primary concern. Intensive testing can facilitate the discovery of bugs, but cannot guarantee their absence. Moreover, developing test suites that grant full coverage of possible behaviours is difficult in the case of reactive systems, due to their above mentioned intrinsic features.

Fuelled by impressive, world fame disaster stories of software failures[1] that (maybe) could have been avoided if formal methods would have been employed, over

[1] Top famous stories include the problems with the Therac 25 radiation therapy engine that in the period 1985-1987 caused the death of several patients by releasing massive overdoses of radiation; the floating-point division bug in the Intel Pentium P5 processor due to an incorrectly coded lookup table and discovered in 1994 by Professor Thomas R. Nicely at Lynchburg College; and the launch failure in Ariane 5.01 maiden flight due to an overflow in data conversion that caused a hardware exception and finally led to self-destruction.

the years, formal methods have provided an extremely useful support in the design of reliable reactive systems and in gaining high confidence that their behaviour will be correct. The application of formal logics and model checking is nowadays common practice in the early and advanced stages of software development, especially in the case of safety-critical industrial applications. While disaster stories do not prove, by themselves, that failures could have been avoided, in the last three decades many success stories can be found in several different areas, such as, e.g., that of mobile communications and security protocols, chip manufacturing, air-traffic control systems, nuclear plants emergency systems.

Formal logics serve to write down unambiguous specifications about how a program is supposed to behave and to reason about system correctness. Classically, we can divide the properties to be investigated in three categories:

safety: properties expressing that something bad will not happen;
liveness: properties expressing that something good will happen;
fairness: properties expressing that something good will happen infinitely often.

The first step in extending HM-logic is to introduce the concept of time, which was present only in a primitive form in the modal operators. This will extend the expressiveness of modal logic, making it able to talk about concepts like "at the next instant of time", "always","never" or "sometimes". When several options are possible, we will also use *path quantifiers*, meaning "for all possible future computations" and "for some possible future computation". In order to represent the concept of time in our logics we have to model it in some mathematical fashion. In our discussion we assume that the time is discrete and infinite.

We start by introducing temporal logics and then present the μ-calculus, which comes equipped with least and greatest fixpoint operators. Notably, most modal and temporal logics can be defined as fragments of the μ-calculus, which in turn provides an elegant and uniform framework for comparison and system verification. Translations from temporal logics to the μ-calculus are of practical relevance, because not only they allow to re-use algorithms for the verification of μ-calculus formulas to check if temporal logics are satisfied, but also because temporal logic formulas are often more readable than specifications written directly in the μ-calculus.

12.2 Temporal Logic

Temporal logic shares similarities with HM-logic, but:

- temporal logic is based on a set of *atomic propositions* whose validity is associated with a set of states, i.e., the observations are taken on states and not on (actions labelling the) arcs;
- temporal operators allow to look further than the "next" operator of HM-logic;
- as we will see, the choice of representing the time as linear (linear temporal logic) or as a tree (computation tree logic) will lead to different types of logic, that roughly correspond to the trace semantic view vs the bisimulation semantics view.

12.2.1 Linear Temporal Logic

In the case of *Linear Temporal Logic* (LTL) the time is represented as a line. This means that the evolutions of the system are linear, they proceed from a state to another without making any choice. The formulas of LTL are based on a set P of *atomic propositions* p, which can be composed using the classical logic operators together with the following temporal operators:

- O: is called *next* operator. The formula $O\phi$ means that ϕ is true in the next state (i.e., in the next instant of time). Some literature uses X or N in place of O.
- F: is called *finally* operator. The formula $F\phi$ means that ϕ is true sometime in the future.
- G: The formula $G\phi$ means that ϕ is always (*globally*) valid in the future.
- U: is called *until* operator. The formula $\phi_0 U \phi_1$ means that ϕ_0 is true until the first time that ϕ_1 is true.

LTL is also called *Propositional Temporal Logic* (PTL).

Definition 12.1 (LTL formulas). The syntax of LTL formulas is defined as follows:

$$\begin{aligned}\phi \ ::= \ & \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid \\ & p \mid O\,\phi \mid F\,\phi \mid G\,\phi \mid \phi_0\, U\, \phi_1\end{aligned}$$

where $p \in P$ is any atomic proposition.

In order to represent the state of the system while the time elapses we introduce the following mathematical structure.

Definition 12.2 (Linear structure). A *linear structure* is a pair (S,P), where P is a set of *atomic propositions* and $S : P \to \wp(\mathbb{N})$ is a function assigning to each proposition $p \in P$ the set of time instants in which it is valid; formally:

$$\forall p \in P.\ S(p) = \{n \in \mathbb{N} \mid n \text{ satisfies } p\}$$

In a linear structure, the natural numbers $0,1,2\ldots$ represent the time instants, and the states in them, and S represents, for every proposition, the states where it holds, or, alternatively, it represents for every state the propositions it satisfies. The temporal operators of LTL allows to quantify (existentially and universally) w.r.t. the traversed states. To define the satisfaction relation, we need to check properties on future states, like some sort of "time travel." To this aim we define the following *shifting* operation on S.

Definition 12.3 (Shifting). Let (S,P) be a linear structure. For any natural number k we let (S^k,P) denote the linear structure where:

$$\forall p \in P.\ S^k(p) = \{n-k \mid n \geq k \ \wedge \ n \in S(p)\}$$

As done for the HM-logic, we define the a notion of satisfaction $\models$ as follows.

Definition 12.4 (LTL satisfaction relation). Given a linear structure (S,P) we define the satisfaction relation $\models$ for LTL formulas by structural induction:

$$
\begin{array}{ll}
S \models \mathit{true} & \\
S \models \neg\phi & \text{if it is not true that } S \models \phi \\
S \models \phi_0 \wedge \phi_1 & \text{if } S \models \phi_0 \text{ and } S \models \phi_1 \\
S \models \phi_0 \vee \phi_1 & \text{if } S \models \phi_0 \text{ or } S \models \phi_1 \\
S \models p & \text{if } 0 \in S(p) \\
S \models O\ \phi & \text{if } S^1 \models \phi \\
S \models F\ \phi & \text{if } \exists k \in \mathbb{N} \text{ such that } S^k \models \phi \\
S \models G\ \phi & \text{if } \forall k \in \mathbb{N} \text{ it holds } S^k \models \phi \\
S \models \phi_0\ U\ \phi_1 & \text{if } \exists k \in \mathbb{N} \text{ such that } S^k \models \phi_1 \text{ and } \forall i < k.\ S^i \models \phi_0
\end{array}
$$

Two LTL formulas ϕ and ψ are called *equivalent*, written $\phi \equiv \psi$ if for any S we have $S \models \phi$ iff $S \models \psi$. From the satisfaction relation it is easy to check that the operators F and G can be expressed in terms of the until operator as follows:

$$
\begin{aligned}
F\ \phi &\equiv \mathit{true}\ U\ \phi \\
G\ \phi &\equiv \neg(F\ \neg\phi)\ \equiv\ \neg(\mathit{true}\ U\ \neg\phi)
\end{aligned}
$$

In the following we let

$$\phi_0 \Rightarrow \phi_1 \stackrel{\text{def}}{=} \phi_1 \vee \neg\phi_0$$

denote the logical implication.

Other commonly used operators are *weak until* (W), *release* (R) and *before* (B). They can be derived as follows:

W: The formula $\phi_0\ W\ \phi_1$ is analogous to the ordinary "until" operator except for the fact that $\phi_0\ W\ \phi_1$ is also true when ϕ_0 holds always, i.e., $\phi_0\ U\ \phi_1$ requires that ϕ_1 holds sometimes in the future, while this is not necessarily the case for $\phi_0\ W\ \phi_1$. Formally, we have:

$$\phi_0\ W\ \phi_1 \stackrel{\text{def}}{=} (\phi_0\ U\ \phi_1) \vee G\ \phi_0$$

R: The formula $\phi_0\ R\ \phi_1$ asserts that ϕ_1 must be true until and including the point where ϕ_0 becomes true. As in the case of weak until, if ϕ_0 never becomes true, then ϕ_1 must hold always. Formally, we have:

$$\phi_0\ R\ \phi_1 \stackrel{\text{def}}{=} \phi_1\ W\ (\phi_1 \wedge \phi_0)$$

B: The formula $\phi_0\ B\ \phi_1$ asserts that ϕ_0 holds sometime before ϕ_1 holds or ϕ_1 never holds. Formally, we have:

$$\phi_0\ B\ \phi_1 \stackrel{\text{def}}{=} \phi_0\ R\ \neg\phi_1$$

We can graphically represent a linear structure S as a diagram like

$$0 \to 1 \to \cdots \to k \to \cdots$$

where additionally each node can be tagged with some of the formulas it satisfies: we write $k_{\phi_1,\ldots,\phi_n}$ if $S^k \models \phi_1 \wedge \cdots \wedge \phi_n$.

For example, given $p, q \in P$, we can visualise the linear structures that satisfy some basic LTL formulas as follows:

$$\begin{array}{rl}
X\ p & 0 \to 1_p \to 2 \to \cdots \\
F\ p & 0 \to \cdots \to (k-1) \to k_p \to (k+1) \to \cdots \\
G\ p & 0_p \to 1_p \to \cdots \to k_p \to \cdots \\
p\ U\ q & 0_p \to 1_p \to \cdots \to (k-1)_p \to k_q \to (k+1) \to \cdots \\
p\ W\ q & \left\{ \begin{array}{l} 0_p \to 1_p \to \cdots \to (k-1)_p \to k_q \to (k+1) \to \cdots \\ 0_p \to 1_p \to \cdots \to k_p \to \cdots \end{array} \right. \\
p\ R\ q & \left\{ \begin{array}{l} 0_q \to 1_q \to \cdots \to (k-1)_q \to k_{p,q} \to (k+1) \to \cdots \\ 0_q \to 1_q \to \cdots \to k_q \to \cdots \end{array} \right. \\
p\ B\ q & \left\{ \begin{array}{l} 0_{\neg q} \to 1_{\neg q} \to \cdots \to (k-1)_{\neg q} \to k_{\neg q,p} \to (k+1) \to \cdots \\ 0_{\neg q} \to 1_{\neg q} \to \cdots \to k_{\neg q} \to \cdots \end{array} \right.
\end{array}$$

We now show some examples that illustrate the expressiveness of LTL.

Example 12.1. Consider the following LTL formulas:

$G\ \neg p$: p will never happen, so it is a safety property.
$p \Rightarrow F\ q$: if p happens now then also q will happen sometime in the future.
$G\ F\ p$: p happens infinitely many times in the future, so it is a fairness property.
$F\ G\ p$: p will hold from some time in the future onward.

Finally, $G(req \Rightarrow (req\ U\ grant))$ expresses the fact that whenever a request is made it holds continuously until it is eventually granted.

12.2.2 Computation Tree Logic

In this section we introduce CTL and CTL*, two logics which use trees as models of time: computation is no longer deterministic along time, but at each instant some possible futures can be taken. CTL and CTL* extend LTL with two operators which allow to express properties on paths over trees. The difference between CTL and CTL* is that the former is a restricted version of the latter. So we start by introducing the more expressive logic CTL*.

12.2.2.1 CTL*

CTL* still includes the temporal operators O, F, G and U: they are called *linear operators*. However, it introduces two new operators, called *path operators*:

- E: The formula $E\ \phi$ (to be read "possibly ϕ") means that there *exists* some path that satisfies ϕ. In the literature it is sometimes written $\exists\ \phi$.
- A: The formula $A\ \phi$ (to be read "inevitably ϕ") means that each path of the tree satisfies ϕ, i.e., that ϕ is satisfied along *all* paths. In the literature it is sometimes written $\forall\ \phi$.

Definition 12.5 (CTL* formulas). The syntax of CTL* formulas is as follows:

$$\begin{aligned}\phi\ ::=\ & \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid \\ & p \mid O\ \phi \mid F\ \phi \mid G\ \phi \mid \phi_0\ U\ \phi_1 \mid \\ & E\ \phi \mid A\ \phi\end{aligned}$$

where $p \in P$ is any atomic proposition.

In the case of CTL*, instead of using linear structures, the computation of the system over time is represented by using infinite trees as explained below.

We recall that a (possibly infinite) tree $T = (V, \rightarrow)$ is a directed graph with vertices in V and directed arcs given by $\rightarrow \subseteq V \times V$, where there is one distinguished vertex $v_0 \in V$ (called *root*) such that there is exactly one directed path from v_0 to any other vertex $v \in V$.

Definition 12.6 (Infinite tree). Let $T = (V, \rightarrow)$ be a tree, with V the set of nodes, v_0 the root and $\rightarrow \subseteq V \times V$ the parent-child relation. We say that T is an *infinite tree* if $\rightarrow$ is *total* on V, namely if every node has a child:

$$\forall v \in V.\ \exists w \in V.\ v \rightarrow w$$

Definition 12.7 (Branching structure). A *branching structure* is a triple (T, S, P), where P is a set of *atomic propositions*, $T = (V, \rightarrow)$ is an infinite tree and $S : P \rightarrow \wp(V)$ is a function from the atomic propositions to subsets of nodes of V defined as follows:

$$\forall p \in P.\ S(p) = \{x \in V \mid x \text{ satisfies } p\}$$

In CTL* computations are described as infinite paths on infinite trees.

Definition 12.8 (Infinite paths). Let $T = (V, \rightarrow)$ be an infinite tree and $\pi = v_0, v_1, \ldots$ be an infinite sequence of nodes in V. We say that π is an *infinite path* over T if

$$\forall i \in \mathbb{N}.\ v_i \rightarrow v_{i+1}$$

Of course, we can view an infinite path $\pi = v_0, v_1, \ldots$ as a function $\pi : \mathbb{N} \rightarrow V$ such that $\pi(i) = v_i$ for any $i \in \mathbb{N}$. As for the linear case, we need a shifting operators on paths.

Definition 12.9 (Path shifting). Let $\pi = v_0, v_1, \ldots$ be an infinite path over T and $k \in \mathbb{N}$. We let the infinite path π^k be defined as follows:

$$\pi^k = v_k, v_{k+1}, \ldots$$

In other words, for an infinite path $\pi : \mathbb{N} \to V$ we let $\pi^k : \mathbb{N} \to V$ be the function defined as $\pi^k(i) = \pi(k+i)$ for all $i \in \mathbb{N}$.

Definition 12.10 (CTL* satisfaction relation). Let (T, S, P) be a branching structure and $\pi = v_0, v_1, v_2, \ldots$ be an infinite path. We define the satisfaction relation $\models$ inductively as follows:

- state operators:

$S, \pi \models \mathit{true}$	
$S, \pi \models \neg\phi$	if it is not true that $S, \pi \models \phi$
$S, \pi \models \phi_0 \wedge \phi_1$	if $S, \pi \models \phi_0$ and $S, \pi \models \phi_1$
$S, \pi \models \phi_0 \vee \phi_1$	if $S, \pi \models \phi_0$ or $S, \pi \models \phi_1$
$S, \pi \models p$	if $v_0 \in S(p)$
$S, \pi \models O\ \phi$	if $S, \pi^1 \models \phi$
$S, \pi \models F\ \phi$	if $\exists i \in \mathbb{N}$ such that $S, \pi^i \models \phi$
$S, \pi \models G\ \phi$	if $\forall i \in \mathbb{N}$ it holds $S, \pi^i \models \phi$
$S, \pi \models \phi_0\ U\ \phi_1$	if $\exists i \in \mathbb{N}$ such that $S, \pi^i \models \phi_1$ and $\forall j < i.\ S, \pi^j \models \phi_0$

- path operators:[2]

$S, \pi \models E\phi$	if there exists $\pi' = v_0, v'_1, v'_2, \ldots$ such that $S, \pi' \models \phi$
$S, \pi \models A\phi$	if for all paths $\pi' = v_0, v'_1, v'_2, \ldots$ we have $S, \pi' \models \phi$

Two CTL* formulas ϕ and ψ are called *equivalent*, written $\phi \equiv \psi$ if for any S, π we have $S, \pi \models \phi$ iff $S, \pi \models \psi$.

Example 12.2. Consider the following CTL* formulas:

$E\ O\ \phi$:	is analogous to the HM-logic formula $\Diamond\phi$.
$A\ G\ p$:	means that p happens in all reachable states.
$E\ F\ p$:	means that p happens in some reachable state.
$A\ F\ p$:	means that on every path there exists a state where p holds.
$E\ (p\ U\ q)$:	means that there exists a path where p holds until q.
$A\ G\ E\ F\ p$:	in every future exists a successive future where p holds.

12.2.2.2 CTL

The formulas of CTL are obtained by restricting CTL*. Let $\{O, F, G, U\}$ be the set of *linear operators*, and $\{E, A\}$ be the set of *path operators*.

[2] Note that in the case of path operators, only the first node v_0 of π is relevant.

Definition 12.11 (CTL formulas). A CTL* formula is a *CTL formula* if all of the followings hold:

1. each path operator appear only immediately before a linear operator;
2. each linear operator appears immediately after a path operator.

In other words, CTL allows only the combined use of path operators with linear operators, like in EO, AO, EF, AF, etc. It is evident that CTL and LTL are both[3] subsets of CTL*, but they are not equivalent to each other. Without going into the detail, we mention that:

- no CTL formula is equivalent to the LTL formula $F\ G\ p$;
- no LTL formula is equivalent to the CTL formula $AG\ (p \Rightarrow (EO\ q \wedge EO\ \neg q))$.

Moreover, fairness is not expressible in CTL.

Finally, we note that all CTL formulas can be written in terms of the minimal set of operators *true*, $\neg$, $\vee$, EG, EU, EO. In fact, for the remaining (combined) operators we have the following logical equivalences:

$$\begin{aligned} EF\phi &\equiv E(true\ U\ \phi) \\ AO\phi &\equiv \neg(EO\neg\phi) \\ AG\phi &\equiv \neg(EF\neg\phi) \equiv \neg E(true\ U\ \neg\phi) \\ AF\phi &\equiv A(true\ U\ \phi) \equiv \neg(EG\neg\phi) \\ A(\phi\ U\ \varphi) &\equiv \neg(E(\neg\varphi\ U\ \neg(\phi\vee\varphi))\vee EG\neg\varphi) \end{aligned}$$

Example 12.3. All the CTL* formulas in Example 12.2 are also CTL formulas.

12.3 μ-Calculus

Now we introduce the μ-calculus. The idea is to add the least and greatest fixpoint operators to modal logic. We remark that HM-logic was introduced not so much as a language to write down system specifications, but rather as an aid to understanding process equivalence from a logical point of view. As a matter of fact, many interesting properties of reactive systems can be conveniently expressed as fixpoints. The two operators that we introduce are the following:

$\mu x.\ \phi$: is the least fixpoint of the equation $x \equiv \phi$.
$\nu x.\ \phi$: is the greatest fixpoint of $x \equiv \phi$.

As a rule of thumb, we can think that least fixpoints are associated with liveness properties, while greatest fixpoints with safety properties.

[3] An LTL formula ϕ is read as the CTL* formula $A\phi$. Namely, the structure where a LTL formula is evaluated corresponds to a CTL* tree consisting of a set of traces.

Definition 12.12 (μ-calculus formulas). The syntax of μ-calculus formulas is:

$$\begin{aligned}\phi \ ::= \ & \mathit{true} \mid \mathit{false} \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid \\ & p \mid \neg p \mid x \mid \Diamond\phi \mid \Box\phi \mid \mu x.\ \phi \mid \nu x.\ \phi\end{aligned}$$

where $p \in P$ is any atomic proposition and $x \in X$ is any predicate variable.

In the following, we let $\mathscr{F}$ denote the set of μ-calculus formulas. To limit the number of parentheses and ease readability of formulas, we tacitly assume that modal operators have higher precedence than logical connectives, and that fixpoint operators have lowest precedence, meaning that the scope of a fixpoint variable extends as far to the right as possible.

The idea is to interpret formulas over a transition system (with vacuous transition labels): to each formula we associate the set of states of the transition system where the formula holds true. Then, the least and greatest fixpoint corresponds quite nicely to the notion of smallest and largest set of states where the formulas holds, respectively.

Since the powerset of the set of states is a complete lattice, in order to apply the fixpoint theory we require that the semantics of any formula ϕ is defined using monotone transformation functions. This is the reason why we do not include general negation in the syntax, but only in the form $\neg p$ for p an atomic proposition. This way, provided that all recursively defined variables are distinct, the μ-calculus formulas we use are said to be in *positive normal form*. Alternatively, we can allow general negation and then require that in well-formed formulas any occurrence of a variable x is preceded by an even number of negations. Then, any such formula can be put in positive normal form by using De Morgan's laws, double negation ($\neg\neg\phi \equiv \phi$) and dualities:

$$\neg\Diamond\phi \equiv \Box\neg\phi \quad \neg\Box\phi \equiv \Diamond\neg\phi \quad \neg\mu x.\ \phi \equiv \nu x.\ \neg\phi[{}^{\neg x}/_{x}] \quad \neg\nu x.\ \phi \equiv \mu x.\ \neg\phi[{}^{\neg x}/_{x}]$$

Let $(V, \to)$ be an LTS (with vacuous transition labels), X be the set of predicate variables and P be a set of propositions, we introduce a function $\rho : P \cup X \to \wp(V)$ which associates to each proposition and to each variable a subset of states of the LTS. Then we define the denotational semantics of μ-calculus which maps each μ-calculus formula ϕ to the subset of states $[\![\phi]\!]\rho$ in which it holds (according to ρ).

Definition 12.13 (Denotational semantics of the μ-calculus). We define the interpretation function $[\![\cdot]\!] : \mathscr{F} \to (P \cup X \to \wp(V)) \to \wp(V)$ by structural recursion on formulas as follows:

$$
\begin{aligned}
[\![\mathit{true}]\!]\rho &= V \\
[\![\mathit{false}]\!]\rho &= \varnothing \\
[\![\phi_0 \wedge \phi_1]\!]\rho &= [\![\phi_0]\!]\rho \cap [\![\phi_1]\!]\rho \\
[\![\phi_0 \vee \phi_1]\!]\rho &= [\![\phi_0]\!]\rho \cup [\![\phi_1]\!]\rho \\
[\![p]\!]\rho &= \rho(p) \\
[\![\neg p]\!]\rho &= V \setminus \rho(p) \\
[\![x]\!]\rho &= \rho x \\
[\![\Diamond\phi]\!]\rho &= \{\, v \mid \exists v' \in [\![\phi]\!]\rho.\ v \to v' \,\} \\
[\![\Box\phi]\!]\rho &= \{\, v \mid \forall v'.\ v \to v' \Rightarrow v' \in [\![\phi]\!]\rho \,\} \\
[\![\mu x.\ \phi]\!]\rho &= \text{fix } \lambda S.\ [\![\phi]\!]\rho[^S/_x] \\
[\![\nu x.\ \phi]\!]\rho &= \text{FIX } \lambda S.\ [\![\phi]\!]\rho[^S/_x]
\end{aligned}
$$

where FIX denotes the greatest fixpoint.

The definitions are straightforward. The only equations that need some comments are those related to the modal operators $\Diamond\phi$ and $\Box\phi$: in the first case, we take as $[\![\Diamond\phi]\!]\rho$ the set of states v that have (at least) one transition to a state v' that satisfies ϕ; in the second case, we take as $[\![\Box\phi]\!]\rho$ the set of states v such that all outgoing transitions lead to some states v' that satisfy ϕ. Note that, as a particular case, a state with no outgoing transitions trivially satisfy the formula $\Box\phi$ for any ϕ. For example the formula $\Box\mathit{false}$ is satisfied by all and only deadlock states; vice versa $\Diamond\mathit{true}$ is satisfied by all and only non-deadlock states. Intuitively, we can note that the modality $\Diamond\phi$ is somewhat analogous to the CTL formula $EO\ \phi$, while the modality $\Box$ can play the role of $AO\ \phi$.

Fixpoints are computed in the $\mathrm{CPO}_\perp$ of sets of states, ordered by inclusion: $(\wp(V), \subseteq)$. Union and intersections are of course monotone functions. Also the functions associated with modal operators

$$\lambda S.\ \{\, v \mid \exists v' \in S.\ v \to v' \,\} \qquad \lambda S.\ \{\, v \mid \forall v'.\ v \to v' \Rightarrow v' \in S \,\}$$

are monotone. The least fixpoint of a function $f : \wp(V) \to \wp(V)$ can then be computed by taking the limit $\bigcup_{n\in\mathbb{N}} f^n(\varnothing)$, while for the greatest fixpoint, we take $\bigcap_{n\in\mathbb{N}} f^n(V)$. In fact, when f is monotone, we have:

$$\varnothing \subseteq f(\varnothing) \subseteq f^2(\varnothing) \subseteq \cdots \subseteq f^n(\varnothing) \subseteq \cdots$$

$$V \supseteq f(V) \supseteq f^2(V) \supseteq \cdots \supseteq f^n(V) \supseteq \cdots$$

Example 12.4 (Basic examples). Let us consider the following formulas:

$\mu x.\ x$: $[\![\mu x.\ x]\!]\rho \stackrel{\text{def}}{=} \text{fix } \lambda S.\ S = \varnothing$.
In fact, let us approximate the result in the usual way:

$$S_0 = \varnothing \qquad S_1 = (\lambda S.\ S)S_0 = S_0$$

$\nu x.\ x$: $[\![\nu x.\ x]\!]\rho \stackrel{\text{def}}{=} \text{FIX}\ \lambda S.\ S = V$.
In fact, we have $S_0 = V$ and $S_1 = (\lambda S.\ S)S_0 = S_0$.

$\mu x.\ \Diamond x$: $[\![\mu x.\ \Diamond x]\!]\rho \stackrel{\text{def}}{=} \text{fix}\ \lambda S.\ \{v \mid \exists v' \in S.\ v \to v'\} = \varnothing$.
In fact, we have:

$$S_0 = \varnothing \qquad S_1 = \{v \mid \exists v' \in \varnothing.\ v \to v'\} = \varnothing.$$

$\mu x.\ \Box x$: $[\![\mu x.\ \Box x]\!]\rho \stackrel{\text{def}}{=} \text{fix}\ \lambda S.\ \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S\}$.
By successive approximations, we get:

$$\begin{aligned}
S_0 &= \varnothing \\
S_1 &= \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in \varnothing\} = \{v \mid v \not\to\} \\
&= \{v \mid v \text{ has no outgoing arc}\} \\
S_2 &= \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_1\} \\
&= \{v \mid v \text{ has outgoing paths of length at most } 1\} \\
&\ldots \\
S_n &= \{v \mid v \text{ has outgoing paths of length at most } n-1\}.
\end{aligned}$$

We can conclude that $[\![\mu x.\ \Box x]\!]\rho = \bigcup_{i\in\mathbb{N}} S_i$ is the set of vertices whose outgoing paths have all finite length.

$\nu x.\ \Box x$: $[\![\nu x.\ \Box x]\!]\rho \stackrel{\text{def}}{=} \text{FIX}\ \lambda S.\ \{v \mid \forall v', v \to v' \Rightarrow v' \in S\} = V$.
In fact, we have:

$$S_0 = V \qquad S_1 = \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in V\} = V.$$

$\mu x.\ p \vee \Diamond x$: $[\![\mu x.\ p \vee \Diamond x]\!]\rho \stackrel{\text{def}}{=} \text{fix}\ \lambda S.\ \rho(p) \cup \{v \mid \exists v' \in S.\ v \to v'\}$.
Let us compute some approximations:

$$\begin{aligned}
S_0 &= \varnothing \\
S_1 &= \rho(p) \\
S_2 &= \rho(p) \cup \{v \mid \exists v' \in \rho(p).\ v \to v'\} \\
&= \{v \mid v \text{ can reach some } v' \in \rho(p) \text{ in less than one step}\} \\
&\ldots \\
S_n &= \{v \mid v \text{ can reach some } v' \in \rho(p) \text{ in less than } n-1 \text{ steps}\} \\
\bigcup_{n\in\mathbb{N}} S_n &= \{v \mid v \text{ has a finite path to some } v' \in \rho(p)\}.
\end{aligned}$$

Thus, the formula is similar to the CTL formula $EF\ p$, meaning that some node in $\rho(p)$ is reachable.

The μ-calculus is more expressive than CTL* (and consequently than CTL and LTL), in fact all CTL* formulas can be translated to μ-calculus formulas. This makes the μ-calculus probably the most studied of all temporal logics of programs.

Unfortunately, the increase in expressive power we get from μ-calculus is balanced in an equally great increase in awkwardness: we invite the reader to check by her/himself how relatively easy is to write down short μ-calculus formulas whose intended meanings remain obscure after several attempts to decipher them. Still, many correctness properties can be expressed in a very concise and elegant way in the μ-calculus. The full translation from CTL* to μ-calculus is quite complex and we do not account for it here.

Example 12.5 (More expressive examples). Let us now briefly discuss some more complicated examples:

$\mu x.\ (p \wedge \Diamond x) \vee q$:	it corresponds to the CTL formula $E(p\ U\ q)$.
$\mu x.\ (p \wedge \Box x \wedge \Diamond x) \vee q$:	it corresponds to the LTL/CTL formula $A(p\ U\ q)$. Note that the sub-formula $\Diamond x$ is needed to discard deadlock states.
$\nu x.\ \mu y.\ (p \wedge \Diamond x) \vee \Diamond y$:	it corresponds to the CTL* formula $EGF\,p$: given a path, $\mu y.\ (p \wedge \Diamond x) \vee \Diamond y$ means that after a finite number of steps you find a vertex where both: (1) p holds, and (2) you can reach a vertex where the property recursively holds.

Without increasing the expressive power of μ-calculus, formulas can be extended to deal with labelled transitions, in the style of HM-logic (see Problem 12.10).

12.4 Model Checking

The problem of model checking consists in the exaustive, possibly automatic, verification of whether a given model of a system meets or not a given logic specification of the properties the system should satisfy, like absence of deadlocks.

The main ingredients of model checking are:

- an LTS M (the model) and a vertex v (the initial state);
- a formula ϕ (in temporal or modal logic) you want to check.

The problem of model checking is: *does v in M satisfy ϕ?*

The result of model checking should be either a positive answer or some counterexample explaining one possible reason why the formula is not satisfied.

Without entering in the details, one successful approach to model checking consists of: 1) computing a finite LTS $M_{\neg\phi}$ that is to some extent equivalent to the negation of the formula ϕ under inspection; roughly, each state in the constructed LTS represents a set of LTL formulas that hold from that state; 2) computing some form of product between the model M and the computed LTS $M_{\neg\phi}$; roughly, this corresponds to solving a non-emptiness problem for the intersection of (the languages associated with) M and $M_{\neg\phi}$; 3) if the intersection is non-empty, then a finite witness can be constructed that offers a counterexample to the validity of the formula ϕ in M.

In the case of μ-calculus formulas, fixpoint theory gives a straightforward (iterative) implementation for a model checker by computing the set of all and only states

that satisfy a formula by successive approximations. In model checking algorithms, it is often convenient to proceed by evaluating formulas with the aid of dynamic programming. The idea is to work in a bottom-up fashion: starting from the atomic predicates that appear in the formula, we mark all the states with the sub-formulas they satisfy. When a variable is encountered, a separate activation of the procedure is allocated for computing the fixpoint of the corresponding recursive definition.

For computing a single fixpoint, the length of the iteration is in general transfinite but is bounded at worst by the cardinal after cardinality of the lattice and in the special case of $\wp(V)$ by the cardinal after the cardinality of V. In practice, many systems can be modelled, at some level of abstraction, as finite state systems, in which case a finite number of iterations ($|V|+1$ at worst) suffices. When two or more fixpoints of the same kind are nested within each other, then we can exploit monotonicity to avoid restarting the computation of the innermost fixpoint at each iteration of the outermost one. However, when least and greatest fixpoints are nested in alternation, this optimisation is no longer possible and the time needed to model check the formula is exponential w.r.t. the so called *alternation depth* of fixpoints in the formula.

From a purely theoretical perspective, the hierarchy obtained by considering formulas ordered according to the alternation depth of fixpoint operators gives more expressive power as the alternation depth increases: model checking in the μ-calculus is proved to be in $\mathsf{NP} \cap \mathsf{coNP}$ (μ-calculus is closed under complementation).

From a pragmatic perspective, any reasonable specification requires at most alternation depth 2 (i.e., it is unlikely to find correctness properties that require alternation depth equal or higher than 3). Moreover, the dominant factor in the complexity of model checking is typically the size of the model rather than the size of the formula, because specifications are often very short: sometimes even exponential growth in the specification size can be tolerable. For these reasons, in many cases, the before mentioned, complex translation from CTL* formulas to μ-calculus formulas is able to guarantee competitive model checking.

In the case of reactive systems, the LTS is often given implicitly, as the one associated with a term of some process algebra, because in this way the structure of the system is handled more conveniently. However, as noted in the previous chapter, even for finite processes, the size of their actual LTS can explode.

When it becomes unfeasible to represent the whole set of states, one approach is to use *abstraction* techniques. Roughly, the idea is to devise a smaller, less detailed model by suppressing inessential data from the original, fully detailed model. Then, as far as the correctness of the larger model follows from the correctness of the smaller model, we are guaranteed that the abstraction is sound.

One possibility to tackle the state explosion problem is to minimise the system according to some suitable equivalence. Note that minimisation can take place also while combining subprocesses and not just at the end. Of course, this technique is viable only if the minimisation preserves all properties to be checked. For example, the validity of any μ-calculus formula is invariant w.r.t. bisimulation, thus we can minimise LTSs up to bisimilarity before model checking them.

Another important technique to succinctly represent large systems is to take a *symbolic* approach, like representing the sets of states where formulas are true in terms of their boolean characteristic functions, expressed as ordered *Binary Decision Diagrams* (BDDs). This approach has been very successful for the debugging and verification of hardware circuits, but, for reasons not well understood, software verification has proved more elusive, probably because programs lack some form of regularity that commonly arises in electronic circuits. In the worst case, also symbolic techniques can lead to intractably inefficient model checking.

Problems

12.1. Suppose there are two processes p_1 and p_2 that can access a single shared resource r. We are given the following atomic propositions, for $i = 1, 2$:

req_i: holds when process p_i is requesting access to r;
use_i: holds when process p_i has had access to r;
rel_i: holds when process p_i has released r.

Use LTL formulas to specify the following properties:

1. mutual exclusion: r is accessed by only one process at a time;
2. release: every time r is accessed by p_i, it is released after a finite amount of time;
3. priority: whenever both p_1 and p_2 require access to r, p_1 is granted access first;
4. absence of starvation: whenever p_i requires access to r, it is eventually granted access to r.

12.2. Consider an elevator system serving three floors, numbered 0 to 2. At each floor there is an elevator door that can be open or closed, a call button, and a light that is on when the elevator has been called. Define a set of atomic propositions, as small as possible, to express the following properties as LTL formulas:

1. a door is not open if the elevator is not present at that floor;
2. every elevator call will be served;
3. every time the elevator serves a floor the corresponding light is turned off;
4. the elevator will always return to floor 0;
5. a request at the top floor has priority over all the other requests.

12.3. Consider the CTL* formula $\phi \stackrel{\text{def}}{=} AF\ G\ (p \vee O\ q)$. Explain the property associated with it and define a branching structure where it is satisfied. Is it a LTL formula? Is it a CTL formula?

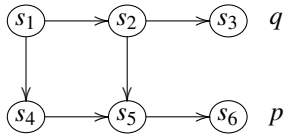
12.4. Prove that if the CTL* formula $AO\ \phi$ is satisfied, then also the formula $O\ A\ \phi$ is satisfied. Is the converse true?

12.5. Is it true that the CTL* formulas $A\ G\ \phi$ and $G\ A\ \phi$ are logically equivalent?

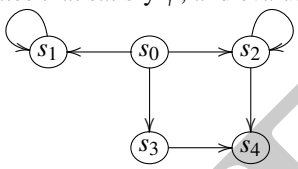
12.6. Given the μ-calculus formula:

$$\phi \stackrel{\text{def}}{=} \nu x.\ (p \vee \Diamond x) \wedge (q \vee \Box x)$$

compute its denotational semantics and evaluate it on the LTS below:



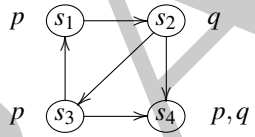
12.7. Given the μ-calculus formula $\phi \stackrel{\text{def}}{=} \nu x.\ \Diamond x$, compute its denotational semantics, spelling out what are the states that satisfy ϕ, and evaluate it on the LTS below:



12.8. Write a μ-calculus formula ϕ representing the statement:

'p is always true along any path leaving the current state.'

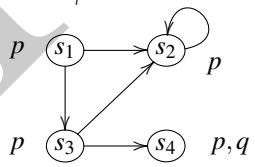
Write the denotational semantics of ϕ and evaluate it over the LTS below:



12.9. Write a μ-calculus formula ϕ representing the statement:

'there is some path where p holds until eventually q holds.'

Write the denotational semantics of ϕ and evaluate it over the LTS below:

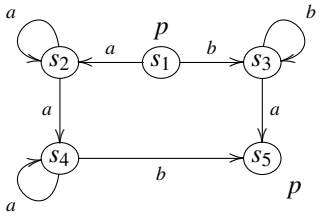


12.10. Let us extend the μ-calculus with the formulas $\langle A \rangle \phi$ and $[A]\phi$, where A is a set of labels: they represent, respectively, the ability to perform a transition with some label $a \in A$ and reach a state that satisfies ϕ, and the necessity to reach a state that satisfies ϕ after performing any transition with label $a \in A$.

1. Define the semantics $[\![\langle A \rangle \phi]\!] \rho$ and $[\![[A]\phi]\!] \rho$.
2. Let us write $\langle a_1, ..., a_n \rangle \phi$ and $[a_1, ..., a_n]\phi$ in place of $\langle \{a_1, ..., a_n\} \rangle \phi$ and $[\{a_1, ..., a_n\}]\phi$, respectively. Compute the denotational semantics of the formulas

$$\phi_1 \stackrel{\text{def}}{=} \nu x.\ ((\langle a\rangle true \wedge \langle b\rangle true) \vee p) \wedge [a,b]x) \qquad \phi_2 \stackrel{\text{def}}{=} \mu x.\ p \vee \langle a,b\rangle x$$

and evaluate them on the LTS below:



Chapter 13
π-Calculus

What's in a name? That which we call a rose by any other name would smell as sweet. (William Shakespeare)

Abstract In this chapter we outline the basic theory of a calculus of processes, called π-calculus. It is not an exaggeration to affirm that π-calculus plays for reactive systems the same foundational role that λ-calculus plays for sequential systems. The key idea is to extend CCS with the ability to send channel names, i.e., π-calculus processes can communicate communication means. The term coined to refer this feature is *name mobility*. The operational semantics of π-calculus is only a bit more involved than that of CCS, while the abstract semantics is considerably more ingenious, because it requires a careful handling of names appearing in the transition labels. In particular, we show that two variants of strong bisimilarity arise naturally, called *early* and *late*, with the former coarser than the latter. We conclude by discussing weak variants of early and late bisimilarities together with compositionality issues.

13.1 Name Mobility

The structures of today's communication systems are not statically defined, but they change continuously according to the needs of the users. The process algebra we have studied in Chapter 11 is unsuitable for modelling such systems, since its communication structure (the channels) cannot evolve dynamically. In this chapter we present the π-calculus, an extension of CCS introduced by Robin Milner, Joachim Parrow and David Walker in 1989, which allows to model mobile systems. The main features of the π-calculus are its ability to create new channel names and to send them in messages, allowing agents to extend their connections. For example, consider the case of the CCS-like process (with value passing)

$$(p \mid q)\backslash a \mid r$$

and suppose that p and q can communicate over the channel a, which is private to them, and that p and r share a channel b for exchanging messages. If we allow

channel names to be sent as message values, then it could be the case that: 1) p sends the name a over the channel b, like in

$$p \stackrel{\text{def}}{=} \overline{b}a.p'$$

for some p'; 2) that q waits for a message on a, like in

$$q \stackrel{\text{def}}{=} a(x).q'$$

for some q' that can exploit x; and 3) that r wants to input a channel name on b, where to send a message m, like in

$$r \stackrel{\text{def}}{=} b(y).\overline{y}m.r'.$$

After the communication between p and r has taken place over the channel b, we would like the scope of a be extended so to include the rightmost process, like in

$$((p' \mid q) \mid \overline{a}m.r'[{}^{a}/_{y}])\backslash a$$

so that q can then input m on a from the process $\overline{a}m.r'$:

$$((p' \mid q'[{}^{m}/_{x}]) \mid r'[{}^{a}/_{y}])\backslash a.$$

All this cannot be achieved in CCS, where restriction is a static operator. Moreover, suppose a process s is initially running in parallel with r, like in

$$(p \mid q)\backslash a \mid (s \mid r).$$

After the communication over b between p and r, we would like the name a to be private to p', q and the continuation of r but not shared by s. Thus if a is already used by s, it must be the case that after the scope extrusion a is renamed to a fresh private name c, not available to s, like in

$$(\,(p'[{}^{c}/_{a}] \mid q[{}^{c}/_{a}]) \mid (s \mid \overline{c}m.r'[{}^{c}/_{y}])\,)\backslash c$$

so that the message $\overline{c}m$ directed to q cannot be intercepted by s.

Remark 13.1 (New syntax for restriction). To differentiate between the static restriction operator of CCS and its dynamic version used in the π-calculus, we write the latter operator in prefix form as $(a)p$ as opposed to the CCS syntax $p\backslash a$. Therefore the initial process of the above example is written

$$(a)(p \mid q) \mid (s \mid r)$$

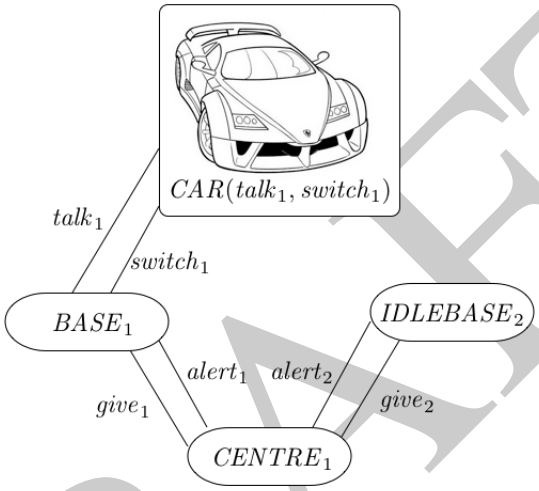
and after the communication it becomes

$$(c)(\,(p'[{}^{c}/_{a}] \mid q[{}^{c}/_{a}]) \mid (s \mid \overline{c}m.r'[{}^{c}/_{y}])\,).$$

The general mechanism for handling name mobility makes the formalisation of the semantics of the π-calculus more complicated than that of CCS, especially because of the side-conditions that serve to guarantee that certain names are fresh.

Let us start with an example which illustrates how the π-calculus can formalise a mobile telephone system.

Example 13.1 (Mobile phones). The following figure represents a mobile phone network: while the car travels, the phone can communicate with different bases in the city, but just one at a time, typically the closest to its position. The communication centre decides when the base must be changed and then the channel for accessing the new base is sent to the car through the switch channel.



As in the dynamic stack Example 11.1 for CCS, also in this case we describe agent behaviour by defining the reachable states:

$$CAR(talk, switch) \stackrel{\text{def}}{=} \overline{talk}.CAR(talk, switch) \; + \; switch(xt, xs).CAR(xt, xs).$$

A car can (recursively) talk on the channel assigned currently by the communication centre (action $\overline{talk}$). Alternatively the car can receive (action $switch(xt, xs)$) a new pair of channels (e.g., $talk'$ and $switch'$) and change the base to which it is connected.

In the example there are two bases, numbered 1 and 2. A generic base $i \in [1,2]$ can be in two possible states: $BASE_i$ or $IDLEBASE_i$.

$$BASE_i \stackrel{\text{def}}{=} talk_i.BASE_i \; + \; give_i(xt, xs).\overline{switch_i}(xt, xs).IDLEBASE_i$$
$$IDLEBASE_i \stackrel{\text{def}}{=} alert_i.BASE_i.$$

In the first case the base is connected to the car, so either the phone can talk or the base can receive two channels from the centre on channel $give_i$, assign them to the variables xt and xs and send them to the car on channel $switch_i$ for allowing it to

change base. In the second case the base i becomes idle, and remains so until it is alerted by the communication centre.

$$CENTRE_1 \stackrel{\text{def}}{=} \overline{give_1}\langle talk_2, switch_2\rangle.\overline{alert_2}.CENTRE_2$$
$$CENTRE_2 \stackrel{\text{def}}{=} \overline{give_2}\langle talk_1, switch_1\rangle.\overline{alert_1}.CENTRE_1.$$

The communication centre can be in different states according to which base is active. In the example there are only two possible states for the communication centre ($CENTRE_1$ and $CENTRE_2$), because only two bases are considered.

Finally we have the process which represents the entire system in the state where the car is talking to the first base.

$$SYSTEM \stackrel{\text{def}}{=} CAR(talk_1, switch_1) \mid BASE_1 \mid IDLEBASE_2 \mid CENTRE_1.$$

Then, suppose that: 1) the centre communicates the names $talk_2$ and $switch_2$ to $BASE_1$ by sending the message $\overline{give_1}\langle talk_2, switch_2\rangle$; 2) the centre alerts $BASE_2$ by sending the message $\overline{alert_2}$; 3) $BASE_1$ tells CAR to switch to channels $talk_2$ and $switch_2$, by sending the message $\overline{switch_i}(talk_2, switch_2)$. Correspondingly, we have:

$$SYSTEM \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} CAR(talk_2, switch_2) \mid IDLEBASE_1 \mid BASE_2 \mid CENTRE_2.$$

Example 13.2 (Secret channel via trusted server). As another example, consider two processes Alice (A) and Bob (B) that want to establish a secret channel using a trusted server (S) with which they already have trustworthy communication link c_{AS} (for Alice to send private messages to the server) and c_{SB} (for the server to send private messages to Bob). The system can be represented by the expression:

$$SYS \stackrel{\text{def}}{=} (c_{AS})(c_{SB})(A \mid S \mid B)$$

where restrictions (c_{AS}) and (c_{SB}) guarantee that channels c_{AS} and c_{SB} are not visible from the environment and where the processes A, S and B are specified as follows:

$$A \stackrel{\text{def}}{=} (c_{AB})\overline{c_{AS}}c_{AB}.\overline{c_{AB}}m.A' \qquad S \stackrel{\text{def}}{=} !c_{AS}(x).\overline{c_{SB}}x.\,\mathbf{nil} \qquad B \stackrel{\text{def}}{=} c_{SB}(y).y(w).B'.$$

Alice defines a private name c_{AB} that wants to use for communicating with B (see the restriction (c_{AB})), then Alice sends the name c_{AB} to the trusted server over their private shared link c_{AS} (output prefix $\overline{c_{AS}}c_{AB}$) and finally sends the message m on the channel c_{AB} (output prefix $\overline{c_{AB}}m$) and continues as A'. The server continuously waits for messages from Alice on channel c_{AS} (input prefix $c_{AS}(x)$) and forwards the content to Bob (output prefix $\overline{c_{SB}}x$). Here the replication operator ! allows to serve multiple requests from Alice by issuing multiple instances of the server process. Bob waits to receive a name to be replaced for y from the server over the channel c_{SB} (input prefix $c_{SB}(y)$) and then uses y to input the message from Alice (input prefix $y(w)$) and then continues as $B'[{}^{c_{AB}}/_{y},{}^{m}/_{w}]$.

13.2 Syntax of the π-calculus

The π-calculus has been introduced to model communicating systems where channel names, representing addresses and links, can be created and forwarded. To this aim we rely on a set of channel names $x, y, z, \ldots$ and extend the CCS actions with the ability to send and receive channel names. In these notes we present the *monadic* version of the calculus, namely the version where names can be sent only one at a time. The *polyadic* version, as used in Example 13.1, is briefly discussed in Problem 13.2.

Definition 13.1 (π-calculus processes). We introduce the π-calculus syntax, with productions for processes p and actions π.

$$\begin{aligned} p &::= \mathbf{nil} \mid \pi.p \mid [x=y]p \mid p+p \mid p|p \mid (y)p \mid !p \\ \pi &::= \tau \mid x(y) \mid \bar{x}y \end{aligned}$$

The meaning of process operators is the following:

nil:	is the inactive agent;
$\pi.p$:	is an agent which can perform an action π and then act like p;
$[x=y]p$:	is the conditional process; it behaves like p if $x=y$, otherwise stays idle;
$p+q$:	is the non-deterministic choice between two processes;
$p\|q$:	is the parallel composition of two processes;
$(y)p$:	denotes the restriction of the channel y with scope p;[1]
$!p$:	is a replicated process: it behaves as if an unbounded number of concurrent occurrences of p were available, all running in parallel. It is the analogous of the (unguarded) CCS recursive process **rec** $x.\ (x\|p)$.

The meaning of the actions π is the following:

τ:	is the invisible action, as usual;
$x(y)$:	is the input on channel x; the received value is stored in y;
$\bar{x}y$:	is the output on channel x of the name y.

In the above cases, we call x the *subject* of the communication (i.e., the channel name where the communication takes place) and y the *object* of the communication (i.e., the channel name that is transmitted or received). As in the λ-calculus, in the π-calculus we have *bound* and *free* occurrence of names. The bounding operators of π-calculus are input and restriction: both in $x(y).p$ and $(y)p$ the name y is bound with scope p. On the contrary, the output prefix is not binding, i.e., if we take the process $\bar{x}y.p$ then the name y is free. Formally, we define the sets of free and bound names of a process by structural recursion as in Figure 13.1. Note that for both $x(y).p$ and $\bar{x}y.p$ the name x is free in p. As usual, we take (abstract) processes up to α-renaming of bound names and write $p[{}^{y}/_{x}]$ for the capture-avoiding substitution of all free-occurrences of the name x with the name y in p.

[1] In the literature the restriction operator is sometimes written $(\nu y)p$ to remark the fact the the name y is "new" to p: we prefer not to use the symbol ν to avoid any conflict with the maximal fixpoint operator, as denoted, e.g., in the μ-calculus (see Chapter 12).

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{fn}(\mathbf{nil}) & \stackrel{\text{def}}{=} & \varnothing & \mathrm{bn}(\mathbf{nil}) & \stackrel{\text{def}}{=} & \varnothing \\
\mathrm{fn}(\tau.p) & \stackrel{\text{def}}{=} & \mathrm{fn}(p) & \mathrm{bn}(\tau.p) & \stackrel{\text{def}}{=} & \mathrm{bn}(p) \\
\mathrm{fn}(x(y).p) & \stackrel{\text{def}}{=} & \{x\} \cup (\mathrm{fn}(p) \setminus \{y\}) & \mathrm{bn}(x(y).p) & \stackrel{\text{def}}{=} & \{y\} \cup \mathrm{bn}(p) \\
\mathrm{fn}(\overline{x}y.p) & \stackrel{\text{def}}{=} & \{x,y\} \cup \mathrm{fn}(p) & \mathrm{bn}(\overline{x}y.p) & \stackrel{\text{def}}{=} & \mathrm{bn}(p) \\
\mathrm{fn}([x=y].p) & \stackrel{\text{def}}{=} & \{x,y\} \cup \mathrm{fn}(p) & \mathrm{bn}([x=y].p) & \stackrel{\text{def}}{=} & \mathrm{bn}(p) \\
\mathrm{fn}(p_0+p_1) & \stackrel{\text{def}}{=} & \mathrm{fn}(p_0) \cup \mathrm{fn}(p_1) & \mathrm{bn}(p_0+p_1) & \stackrel{\text{def}}{=} & \mathrm{bn}(p_0) \cup \mathrm{bn}(p_1) \\
\mathrm{fn}(p_0|p_1) & \stackrel{\text{def}}{=} & \mathrm{fn}(p_0) \cup \mathrm{fn}(p_1) & \mathrm{bn}(p_0|p_1) & \stackrel{\text{def}}{=} & \mathrm{bn}(p_0) \cup \mathrm{bn}(p_1) \\
\mathrm{fn}((y).p) & \stackrel{\text{def}}{=} & \mathrm{fn}(p) \setminus \{y\} & \mathrm{bn}((y).p) & \stackrel{\text{def}}{=} & \{y\} \cup \mathrm{bn}(p) \\
\mathrm{fn}(!p) & \stackrel{\text{def}}{=} & \mathrm{fn}(p) & \mathrm{bn}(!p) & \stackrel{\text{def}}{=} & \mathrm{bn}(p)
\end{array}
$$

Fig. 13.1: Free names and bound names of processes

Unlike for CCS, the scope of the name y in the restricted process $(y)p$ can be dynamically extended to include other processes than p by name extrusion. The possibility to enlarge the scope of a restricted name is a very useful, intrinsic feature of the π-calculus. In fact, in the π-calculus, channel names are data that can be transmitted, so the process p can send the name private y to another process q which thus falls under the scope of y (see Section 13.1). Name extrusion allows us to modify the structure of private communications between agents. Moreover, it is a convenient way to formalise secure data transmission, as implemented, e.g., via cryptographic protocols.

13.3 Operational Semantics of the π-calculus

We define the operational semantics of the π-calculus by deriving an LTS via inference rules. Well-formed formulas are written $p \xrightarrow{\alpha} q$ for suitable processes p, q and label α. The syntax of labels is richer than the one used in the case of CCS, as defined next.

Definition 13.2 (Action labels). The possible actions α that label the transitions are:

τ: the silent action;
$x(y)$: the input of a fresh name y on channel x;
$\overline{x}y$: the free output of name y on channel x;
$\overline{x}(y)$: the bound output (called *name extrusion*) of a restricted name y on channel x.

The definition of free names $\mathrm{fn}(\cdot)$ and bound names $\mathrm{bn}(\cdot)$ are extended to labels as defined in Figure 13.2. Moreover, we let $\mathrm{n}(\alpha) \stackrel{\text{def}}{=} \mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha)$ denote the set of all names appearing in α.

Most transitions $p \xrightarrow{\alpha} q$ can be read as the computational evolution of p to q when the action α is performed, analogously to the ones for CCS. The only exceptions are input-labelled transitions: if $p \xrightarrow{x(y)} p'$ for some x and (fresh) y, then the computational

$$\begin{array}{rlrl} \mathrm{fn}(\tau) & \stackrel{\text{def}}{=} \varnothing & \mathrm{bn}(\tau) & \stackrel{\text{def}}{=} \varnothing \\ \mathrm{fn}(x(y)) & \stackrel{\text{def}}{=} \{x\} & \mathrm{bn}(x(y)) & \stackrel{\text{def}}{=} \{y\} \\ \mathrm{fn}(\overline{x}y) & \stackrel{\text{def}}{=} \{x,y\} & \mathrm{bn}(\overline{x}y) & \stackrel{\text{def}}{=} \varnothing \\ \mathrm{fn}(\overline{x}(y)) & \stackrel{\text{def}}{=} \{x\} & \mathrm{bn}(\overline{x}(y)) & \stackrel{\text{def}}{=} \{y\} \end{array}$$

Fig. 13.2: Free names and bound names of labels

evolution of p depends on the actual received name z to be substituted for y in p', but the input transition is just given for a generic formal parameter y, not for all its possible instances. For example, it may well be the case that one of the free names of p is received, while y stands just for fresh names. The main consequence is that, in the bisimulation game, the attacker can pick a received name z and the defender must choose an input transition $q \xrightarrow{x(y)} q'$ such that $p'[{}^{z}/_{y}]$ and $q'[{}^{z}/_{y}]$ are related (not p' and q'). Depending on the moment when the name is chosen by the attacker, before or after the move of the defender, two different notions of bisimulation arise, as explained in Section 13.5.

We can now present the inference rules for the operational semantics of the π-calculus and briefly comment on them.

13.3.1 Inactive Process

As in the case of CCS, there is no rule for the inactive process **nil**: it has no outgoing transition.

13.3.2 Action Prefix

There are three rules for an action prefixed process $\pi.p$, one for each possible shape of the prefix π.

$$\text{(Tau)}\ \frac{}{\tau.p \xrightarrow{\tau} p}$$

The rule (Tau) allows to perform invisible actions.

$$\text{(Out)}\ \frac{}{\overline{x}\,y.p \xrightarrow{\overline{x}\,y} p}$$

As we said, the π-calculus processes can exchange messages which can contain information (i.e., channel names). The rule (Out) allows a process to send the name y on the channel x.

$$\text{(In)}\ \frac{}{x(y).p \xrightarrow{x(w)} p[^{w}/_{y}]}\ w \notin \mathrm{fn}((y)p)$$

The rule (In) allows to receive in input over x some channel name. The label $x(w)$ records that some formal name w is received, which is substituted for y in the continuation process p. In order to avoid name clashes, we assume w does not appear as a free name in $(y)p$, i.e., the transition is defined only when w is *fresh*. Of course, as a special case, w can be y. The side-condition may appear unacceptable, as possibly known names could be received, but this is convenient to express two different kinds of abstract semantics over the same LTS, as we will discuss later in Sections 13.5.1 and 13.5.2. For example, we have the transitions

$$x(y).\overline{y}z.\,\mathbf{nil} \xrightarrow{x(w)} \overline{w}z.\,\mathbf{nil} \xrightarrow{\overline{w}z} \mathbf{nil}$$

but we do not have the transition (because $z \in \mathrm{fn}((y)\overline{y}z.\,\mathbf{nil})$)

$$x(y).\overline{y}z.\,\mathbf{nil} \not\xrightarrow{x(z)} \overline{z}z.\,\mathbf{nil}.$$

Remark 13.2. The rule (In) introduces an infinite branching, because there are infinitely many fresh names w that can be substituted for y. One could try to improve the situation by choosing a standard representative, but such a representative cannot be unique for all contexts (see Problem 13.10). Another possibility is to introduce a special symbol, say $\bullet$, to denote that in the continuation $p[^{\bullet}/_{y}]$ (no longer a π-calculus process) some actual argument should be provided.

13.3.3 Name Matching

$$\text{(Match)}\ \frac{p \xrightarrow{\alpha} p'}{[x=x]p \xrightarrow{\alpha} p'}$$

The rule (Match) allows to check the equality of names before releasing p. If the matching condition is not satisfied the execution halts. Name matching can be used to write a process that receives a name and then tests this name to choose what to do next. For example, a login process for an account whose password is *pwd* could be written $\mathit{login}(y).[y=\mathit{pwd}]p$.

13.3.4 Choice

$$\text{(SumL)}\ \frac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'} \qquad \text{(SumR)}\ \frac{q \xrightarrow{\alpha} q'}{p+q \xrightarrow{\alpha} q'}$$

The rules (SumL) and (SumR) allow the system $p+q$ to behave as either p or q. They are completely analogous to the rules for choice in CCS.

13.3.5 Parallel Composition

There are six rules for parallel composition. Here we present the first four. The remaining two rules deal with name extrusion and are presented in Section 13.3.7.

$$\text{(ParL)}\ \frac{p \xrightarrow{\alpha} p'}{p \mid q \xrightarrow{\alpha} p' \mid q}\ \mathrm{bn}(\alpha) \cap \mathrm{fn}(q) = \varnothing \qquad \text{(ParR)}\ \frac{q \xrightarrow{\alpha} q'}{p \mid q \xrightarrow{\alpha} p \mid q'}\ \mathrm{bn}(\alpha) \cap \mathrm{fn}(p) = \varnothing$$

As for CCS, the two rules (ParL) and (ParR) allow the interleaved execution of two π-calculus agents. The side conditions guarantee that the bound names in α (if any) are fresh w.r.t. the idle process. For example, a valid transition is

$$x(y).\overline{y}z.\,\mathbf{nil} \mid w(u).\,\mathbf{nil} \xrightarrow{x(v)} \overline{v}z.\,\mathbf{nil} \mid w(u).\,\mathbf{nil}.$$

Instead, we do not allow the transition

$$x(y).\overline{y}z.\,\mathbf{nil} \mid w(u).\,\mathbf{nil} \not\xrightarrow{x(w)} \overline{w}z.\,\mathbf{nil} \mid w(u).\,\mathbf{nil}$$

because the received name $w \in \mathrm{bn}(x(w))$ clashes with the free name $w \in \mathrm{fn}(w(u).\,\mathbf{nil})$.

$$\text{(ComL)}\ \frac{p \xrightarrow{\overline{x}z} p' \quad q \xrightarrow{x(y)} q'}{p \mid q \xrightarrow{\tau} p' \mid (q'[^{z}/_{y}])} \qquad \text{(ComR)}\ \frac{p \xrightarrow{x(y)} p' \quad q \xrightarrow{\overline{x}z} q'}{p \mid q \xrightarrow{\tau} p'[^{z}/_{y}] \mid q'}$$

The rules (ComL) and (ComR) allow the synchronisation of two parallel processes. The formal name y is replaced with the actual name z in the continuation of the receiver. For example, we can derive the transition

$$x(y).\overline{y}z.\,\mathbf{nil} \mid \overline{x}z.y(v).\,\mathbf{nil} \xrightarrow{\tau} \overline{z}z.\,\mathbf{nil} \mid y(v).\,\mathbf{nil}$$

13.3.6 Restriction

$$\text{(Res)}\ \frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'}\ y \notin \mathrm{n}(\alpha)$$

The rule (Res) expresses the fact that if a name y is restricted on top of the process p, then any action that does not involve y can be performed by p.

13.3.7 Scope Extrusion

Now we present the most important rules of π-calculus, (Open) and (Close), dealing with *scope extrusion* of channel names. Rule (Open) makes public a private channel name, while rule (Close) restricts again the name, but with a broader scope.

$$\text{(Open)}\ \frac{p \xrightarrow{\overline{x}\,y} p'}{(y)p \xrightarrow{\overline{x}(w)} p'[{}^{w}/_{y}]}\ y \neq x \wedge w \notin \mathrm{fn}((y)p)$$

The rule (Open) publishes the private name w, which is guaranteed to be fresh. Of course, as a special case, we can take $w = y$.

Remark 13.3. The rule (Open), as the rule (In), introduces an infinite branching, because there are infinitely many fresh names w that can be taken. The main difference is that, in the bisimulation game, when the move $p \xrightarrow{\overline{x}(w)} p'$ of the attacker is matched by the move $q \xrightarrow{\overline{x}(w)} q'$ of the defender, then p' and q' must be directly related, i.e., it is not necessary to check that $p'[{}^{z}/_{w}]$ and $q'[{}^{z}/_{w}]$ are related for any z, because extruded names must be fresh and all fresh names are already accounted for by bound output transitions.

$$\text{(CloseL)}\ \frac{p \xrightarrow{\overline{x}\,(w)} p' \quad q \xrightarrow{x(w)} q'}{p \mid q \xrightarrow{\tau} (w)(p' \mid q')} \qquad \text{(CloseR)}\ \frac{p \xrightarrow{x(w)} p' \quad q \xrightarrow{\overline{x}\,(w)} q'}{p \mid q \xrightarrow{\tau} (w)(p' \mid q')}$$

The rules (CloseL) and (CloseR) transform the object w of the communication over x in a private channel between p and q. Freshness of w is guaranteed by rules (In), (Open), (ParL) and (ParR). For example, we have

$$x(y).\overline{y}z.\mathbf{nil} \mid (z)\overline{x}z.z(y).\mathbf{nil} \xrightarrow{\tau} (u)(\overline{u}z.\mathbf{nil} \mid u(y).\mathbf{nil}).$$

13.3.8 Replication

$$\text{(Rep)}\ \frac{p \mid !p \xrightarrow{\alpha} p'}{!p \xrightarrow{\alpha} p'}$$

The last rule deals with replication. It allows to replicate a process as many times as needed, in a reentrant fashion, without consuming it. Notice that $!p$ is able also to perform the synchronisations between two copies of p, if possible at all.

13.3.9 A Sample Derivation

Example 13.3 (Scope extrusion). We conclude this section by showing an example of the use of the rule system. Let us consider the following system:

$$(((y)\bar{x}y.p) \mid q) \mid x(z).r$$

where p, q, r are π-calculus processes. The process $(y)\bar{x}\,y.p$ would like to set up a private channel with $x(z).r$, which however should remain hidden to q. By using the inference rules of the operational semantics we can proceed in a goal-oriented fashion to find a derivation for the corresponding transition:

$$
\begin{array}{l}
(((y)\bar{x}y.p) \mid q) \mid x(z).r \xrightarrow{\alpha} s \\
\quad \nwarrow_{\text{(CloseL)},\ \alpha=\tau,\ s=(w)(s_1 \mid r_1)} \quad ((y)\bar{x}y.p) \mid q \xrightarrow{\bar{x}(w)} s_1, \quad x(z).r \xrightarrow{x(w)} r_1 \\
\quad\quad \nwarrow_{\text{(ParL)},\ s_1=p_1 \mid q,\ w\notin \mathrm{fn}(q)} \quad (y)\bar{x}y.p \xrightarrow{\bar{x}(w)} p_1 \quad x(z).r \xrightarrow{x(w)} r_1 \\
\quad\quad\quad \nwarrow_{\text{(Open)},\ p_1=p_2[^w/_y],\ w\notin \mathrm{fn}((y).p)} \quad \bar{x}y.p \xrightarrow{\bar{x}y} p_2, \quad x(z).r \xrightarrow{x(w)} r_1 \\
\quad\quad\quad\quad \nwarrow^{*}_{\text{(Out)+(In)},\ r_1=r[^w/_z],\ p_2=p,\ w\notin \mathrm{fn}((z).r)} \quad \square
\end{array}
$$

so we have:

$$
\begin{aligned}
p_2 &= p \\
p_1 &= p_2[^w/_y] = p[^w/_y] \\
r_1 &= r[^w/_z] \\
s_1 &= p_1 \mid q = p[^w/_y] \mid q \\
s &= (w)(s_1 \mid r_1) = (w)\,(\,(p[^w/_y] \mid q) \mid (r[^w/_z])\,) \\
\alpha &= \tau
\end{aligned}
$$

In conclusion:

$$(((y)\bar{x}y.p) \mid q) \mid x(z).r \xrightarrow{\tau} (w)\,(\,(p[^w/_y] \mid q) \mid (r[^w/_z])\,)$$

under the condition that w is fresh, i.e., that $w \notin \mathrm{fn}(q) \cup \mathrm{fn}((y)p) \cup \mathrm{fn}((z)r)$.

13.4 Structural Equivalence of π-calculus

As we have already noticed for CCS, there are different terms representing essentially the same process. As the complexity of the calculus increases, it is more and more convenient to manipulate terms up to some intuitive structural axioms. In the following we denote by $\equiv$ the least congruence[2] over π-calculus processes that includes

[2] This means that $\equiv$ is reflexive, symmetric, transitive and closed under context embedding.

$$
\begin{array}{lll}
p + \mathbf{nil} \equiv p & p + q \equiv q + p & (p+q)+r \equiv p+(q+r) \\
p \mid \mathbf{nil} \equiv p & p \mid q \equiv q \mid p & (p \mid q) \mid r \equiv p \mid (q \mid r) \\
(x)\,\mathbf{nil} \equiv \mathbf{nil} & (y)(x)p \equiv (x)(y)p & (x)(p \mid q) \equiv p \mid (x)q \ \text{ if } \ x \notin \mathrm{fn}(p) \\
[x=y]\,\mathbf{nil} \equiv \mathbf{nil} & [x=x]\,p \equiv p & p \mid !p \equiv !p
\end{array}
$$

Fig. 13.3: Axioms for structural equivalence

α-conversion of bound names and that is induced by the set of axioms in Figure 13.3. The relation $\equiv$ is called *structural equivalence*.

13.4.1 Reduction semantics

The operational semantics of π-calculus is much more complicated than that of CCS because it needs to handle name passing and scope extrusion. By exploiting structural equivalence we can define a so-called *reduction semantics* that is simpler to understand. The idea is to define an LTS with silent labels only, that models all the interactions that can take place in a process, without considering interactions with the environment. This is accomplished by first rewriting the process to a structurally equivalent normal form and then by applying basic reduction rules. In fact it can be proved that for each π-calculus process p there exists:

- a finite number of names $x_1, x_2, ..., x_k$;
- a finite number of guarded sums[3] $s_1, s_2, ..., s_n$;
- and a finite number of processes $p_1, p_2, ..., p_m$, such that

$$P \equiv (x_1) \cdots (x_k)(s_1 \mid \cdots \mid s_n \mid !p_1 \mid \cdots \mid !p_m)$$

Then, a reduction is either a silent action performed by some s_i or a communication from an input prefix of say s_i with an output prefix of say s_j. We write the reduction relation as a binary relation on processes using the notation $p \mapsto q$ for indicating that p reduces to q in one step. The rules defining the relation $\mapsto$ are the following:

$$
\frac{}{\tau.p + s \mapsto p} \qquad \frac{}{(x(y).p_1 + s_1) \mid (\overline{x}z.p_2 + s_2) \mapsto p_1[z/y] \mid p_2}
$$

$$
\frac{p \mapsto p'}{p \mid q \mapsto p' \mid q} \qquad \frac{p \mapsto p'}{(x)p \mapsto (x)p'} \qquad \frac{p \equiv q \quad q \mapsto q' \quad q' \equiv p'}{p \mapsto p'}
$$

The reduction semantics can be put in correspondence with the (silent transitions of the) labelled operational semantics by the following theorem.

Lemma 13.1 (Harmony Lemma). *For any π-calculus processes p, p' and any action α we have that:*

[3] They are non-deterministic choices whose arguments are action prefixed processes, i.e., they take the form $\pi_1.p_1 + \cdots + \pi_h.p_h$.

1. $\exists q.\ p \equiv q \xrightarrow{\alpha} p'$ *implies that* $\exists q'.\ p \xrightarrow{\alpha} q' \equiv p'$
2. $p \mapsto p'$ *if and only if* $\exists q'.\ p \xrightarrow{\tau} q' \equiv p'$.

Proof. We only sketch the proof.

1. The first fact can be proved by showing that the thesis holds for each single application of any structural axiom and then proving the general case by mathematical induction on the length of the proof of structural equivalence of p and q.
2. The second fact requires to prove the two implications separately:

   $\Rightarrow$) We prove first that, if $p \mapsto p'$, then we can find equivalent processes $r \equiv p$ and $r' \equiv p'$ in suitable form, such that $r \xrightarrow{\tau} r'$. Finally, from $p \equiv r \xrightarrow{\tau} r'$ we conclude by the first fact that $\exists q' \equiv r'$ such that $p \xrightarrow{\tau} q'$, since $q' \equiv p'$ by transitivity of $\equiv$.

   $\Leftarrow$) After showing that, for any p, q, whenever $p \xrightarrow{\alpha} q$ then we can find suitable processes $p' \equiv p$ and $q' \equiv q$ in normal form, we prove, by rule induction on $p \xrightarrow{\tau} p'$, that for any p, p', if $p \xrightarrow{\tau} p'$, then $p \mapsto p'$, from which the thesis follows immediately. □

13.5 Abstract Semantics of the π-calculus

Now we present an abstract semantics of π-calculus, namely we disregard the syntax of processes but focus on their behaviours. As we saw in CCS, one of the main goals of abstract semantics is to find the correct degree of abstraction, depending on the properties that we want to study. Thus also in this case there are many kinds of bisimulations that lead to different bisimilarities, which are useful in different circumstances.

We start from *strong bisimulation* of π-calculus, which is an extended version of the strong bisimulation of CCS, here complicated by the side-conditions on bound names of actions and by the fact that, after an input, we want the continuation processes to be equivalent for any received name. An important new feature of π-calculus is the choice of the time when the names used as objects of input transitions are assigned their actual values. If they are assigned *before* the choice of the (bi)simulating transition, namely if the choice of the transition may depend on the assigned value, we get the *early* bisimulation. Instead, if the choice must hold for all possible names, we have the *late* bisimulation case. As we will see in short, the second option leads to a finer semantics. Finally, we will present the *weak bisimulation* for π-calculus. In all the above cases, the congruence property is not satisfied by the largest bisimulations, so that the equivalences must be closed under suitable contexts to get the corresponding observational congruences.

13.5.1 Strong Early Ground Bisimulations

In *early* bisimulation we require that for each name w that an agent can receive on a channel x there exists a state q' in which the bisimilar agent will be after receiving w on x. This means that the bisimilar agent can choose a different transition (and thus a different state q') depending on the observed name w.

Formally, a binary relation S on π-calculus agents is a *strong early ground bisimulation* if:

$$\forall p,q.\ p\,S\,q \Rightarrow \begin{cases} \forall p'. & \text{if } p \xrightarrow{\tau} p' & \text{then } \exists q'.\ q \xrightarrow{\tau} q' \text{ and } p'\,S\,q' \\ \forall x,y,p'. & \text{if } p \xrightarrow{\bar{x}y} p' & \text{then } \exists q'.\ q \xrightarrow{\bar{x}y} q' \text{ and } p'\,S\,q' \\ \forall x,y,p'. & \text{if } p \xrightarrow{\bar{x}(y)} p' & \text{with } y \notin \mathrm{fn}(q), \\ & & \text{then } \exists q'.\ q \xrightarrow{\bar{x}(y)} q' \text{ and } p'\,S\,q' \\ \forall x,y,p'. & \text{if } p \xrightarrow{x(y)} p' & \text{with } y \notin \mathrm{fn}(q), \\ & & \text{then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'[{}^{w}/_{y}]\,S\,q'[{}^{w}/_{y}] \\ \text{(and vice versa)} & & \end{cases}$$

Of course, "vice versa" means that other four cases are present, where q challenges p to (bi)simulate its transitions. Note that in the case of silent label τ or output labels $\bar{x}y$ the definition of bisimulation is as expected. The case of bound output labels $\bar{x}(y)$ has the additional condition $y \notin \mathrm{fn}(q)$ as it makes sense to consider only moves where y is fresh for both p and q.[4] The more interesting case is that of input labels $x(y)$: here we have the same condition $y \notin \mathrm{fn}(q)$ as in the case of bound output (for exactly the same reason), but additionally we require that p' and q' are compared w.r.t. all possible received names $p'[{}^{w}/_{y}]\,S\,q'[{}^{w}/_{y}]$. Notice that, as obvious for a generic input, also names which are not fresh (namely that appear free in p' and q') can replace variable y. This is the reason why we required y to be fresh in the first place. It is important to remark that different moves of q can be chosen depending on the received value w: this is the main feature of *early* bisimilarity.

The very same definition of strong early ground bisimulation can be written more concisely by grouping together the three cases of silent label, output labels and bound output labels in the same clause:

$$\forall p,q.\ p\,S\,q \Rightarrow \begin{cases} \forall \alpha,p'. & \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y) \wedge \mathrm{bn}(\alpha) \cap \mathrm{fn}(q) = \varnothing, \\ & \quad \text{then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\,S\,q' \\ \forall x,y,p'. & \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin \mathrm{fn}(q), \\ & \quad \text{then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'[{}^{w}/_{y}]\,S\,q'[{}^{w}/_{y}] \\ \text{(and vice versa)} & \end{cases}$$

[4] In general, a bisimulation can relate processes whose sets of free names are different, as they are not necessarily used. For example, we want to relate p and $p \mid q$ when q is deadlocked, even if $\mathrm{fn}(q) \neq \varnothing$, so the condition $y \notin \mathrm{fn}(p \mid q)$ is necessary to allow $p \mid q$ to (bi)simulate all bound output moves of p, if any.

Remark 13.4. While the second clause introduces universal quantification over the received names, it is enough to check that the condition $p'[^w/_y]\ S\ q'[^w/_y]$ is satisfied for all $w \in \text{fn}(p') \cup fn(q')$ and for a *single* fresh name $w \notin \text{fn}(p') \cup fn(q')$, i.e., for a finite set of names.

Definition 13.3 (Early bisimilarity $\sim_E$). Two π-calculus agents p and q are *early bisimilar*, written $p \sim_E q$, if there exists a strong early ground bisimulation S such that $p\ S\ q$.

Example 13.4 (Early bisimilar processes). Let us consider the processes:

$$p \stackrel{\text{def}}{=} x(y).\tau.\mathbf{nil} + x(y).\mathbf{nil} \qquad q \stackrel{\text{def}}{=} p + x(y).[y=z]\tau.\mathbf{nil}$$

whose transitions are (for any fresh name u):

$$\begin{array}{ll} p \xrightarrow{x(u)} \tau.\mathbf{nil} & q \xrightarrow{x(u)} \tau.\mathbf{nil} \\ p \xrightarrow{x(u)} \mathbf{nil} & q \xrightarrow{x(u)} \mathbf{nil} \\ & q \xrightarrow{x(u)} [u=z]\tau.\mathbf{nil} \end{array}$$

The two processes p and q are early bisimilar. On the one hand, it is obvious that q can simulate all moves of p. On the other hand, let q perform an input operation on x by choosing the rightmost option. Then, we need to find, for each received name w to be substituted for u, a transition $p \xrightarrow{x(u)} p'$ such that $p'[^w/_u]$ is early bisimilar to $[w=z]\tau.\mathbf{nil}$. If the received name is $w=z$, then the match is satisfied and p can choose to perform the left input operation to reach the state $\tau.\mathbf{nil}$, which is early bisimilar to $[z=z]\tau.\mathbf{nil}$. Otherwise, if $w \neq z$, then the match condition is not satisfied and $[w=z]\tau.\mathbf{nil}$ is deadlock, so p can choose to perform the right input operation and reach the deadlock state **nil**. Notably, in the early bisimulation game, the received name is known prior to the choice of the transition by the defender.

13.5.2 Strong Late Ground Bisimulations

In the case of late bisimulation, we require that, if an agent p has an input transition to p', then there exists a *single* input transition of q to q' such that p' and q' are related *for any* received value, i.e., q must choose the transition without knowing what the received value will be.

Formally, a binary relation S on π-calculus agents is a *strong late ground bisimulation* if (in concise form):

$$\forall p,q.\ p\ S\ q\ \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{ if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y) \wedge \mathrm{bn}(\alpha) \cap \mathrm{fn}(q) = \varnothing, \\ \qquad \text{then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{ if } p \xrightarrow{x(y)} p' \text{ with } y \notin \mathrm{fn}(q), \\ \qquad \text{then } \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } \forall w.\ p'[^w/_y]\ S\ q'[^w/_y] \\ \text{(and vice versa)} \end{cases}$$

The only difference w.r.t. the definition of strong early ground bisimulation is that, in the second clause, the order of quantifiers $\exists q'$ and $\forall w$ is inverted.

Remark 13.5. In the literature, early and late bisimulations are often defined over two different transition systems. For example, if only early bisimilarity is considered, then the labels for input transitions could contain the actual received name, which can be either free or fresh. We have chosen to define a single transition system to give an uniform presentation of the two abstract semantics.

Definition 13.4 (Late bisimilarity $\sim_L$). Two π-calculus agents p and q are said to be *late bisimilar*, written $p \sim_L q$ if there exists a strong late ground bisimulation S such that $p\ S\ q$.

The next example illustrates the difference between late and early bisimilarities.

Example 13.5 (Early vs late bisimulation). Let us consider again the early bisimilar processes p and q from Example 13.3. When late bisimilarity is considered, then the two agents are not equivalent. In fact p should find a state which can handle all the possible names received on x. If the leftmost choice is selected, then $\tau.\mathbf{nil}$ is equivalent to $[w = z].\tau.\mathbf{nil}$ only when when the received value $w = z$ but not in the other cases. On the other hand, if the right choice is selected, then $\tau.\mathbf{nil}$ is equivalent to $[w = z].\tau.\mathbf{nil}$ only when $w \neq z$.

As the above example suggests, it is possible to prove that early bisimilarity is strictly coarser than late: if p and q are late bisimilar, then they are early bisimilar.

13.5.3 Compositionality and Strong Full Bisimilarities

Unfortunately both early and late ground bisimilarities are not congruences, even in the strong case, as shown by the following counterexample.

Example 13.6 (Ground bisimilarities are not congruences). Let us consider the following agents:

$$p \stackrel{\text{def}}{=} \bar{x}x.\mathbf{nil} \mid x'(y).\mathbf{nil} \qquad q \stackrel{\text{def}}{=} \bar{x}x.x'(y).\mathbf{nil} + x'(y).\bar{x}x.\mathbf{nil}$$

We leave the reader to check that the agents p and q are bisimilar (according to both early and late bisimilarities). Now, in order to show that ground bisimulations are not congruences, we define the following context:

$$C[\cdot] = z(x').[\cdot]$$

by plugging p and q inside the hole of $C[\cdot]$ we get:

$$C[p] = z(x').(\bar{x}x.\mathbf{nil} \mid x'(y).\mathbf{nil}) \qquad C[q] = z(x').(\bar{x}x.x'(y).\mathbf{nil} + x'(y).\bar{x}x.\mathbf{nil})$$

$C[p]$ and $C[q]$ are not early bisimilar (and thus not late bisimilar). In fact, suppose the name x is received on z: we need to compare the agents

$$p' \stackrel{\text{def}}{=} \bar{x}x.\mathbf{nil} \mid x(y).\mathbf{nil} \qquad q' \stackrel{\text{def}}{=} \bar{x}x.x(y).\mathbf{nil} + x(y).\bar{x}x.\mathbf{nil}$$

Now p' can perform a τ-transition, but q' cannot.

The problem illustrated by the previous example is due to aliasing, and it appears often in programming languages with both global variables and parameter passing to procedures. It can be solved by defining a finer relation between agents called *strong early full bisimilarity* and defined as follows:

$$p \simeq_E q \quad \Leftrightarrow \quad p\sigma \sim_E q\sigma \text{ for every substitution } \sigma$$

where a substitution σ is a function from names to names that is equal to the identity function almost everywhere (i.e., it differs from the identity function only on a finite number of elements of the domain).

Analogously, we can define *strong late full bisimilarity* $\simeq_L$ by letting

$$p \simeq_L q \quad \Leftrightarrow \quad p\sigma \sim_L q\sigma \text{ for every substitution } \sigma$$

13.5.4 Weak Early and Late Ground Bisimulations

As for CCS, we can define the weak versions of transitions $\stackrel{\alpha}{\Rightarrow}$ and of bisimulation relations. The definition of weak transitions is the same as CCS: 1) we write $p \stackrel{\tau}{\Rightarrow} q$ if p can reach q via a, possibly empty, sequence of τ-transitions; and 2) we write $p \stackrel{\alpha}{\Rightarrow} q$ for $\alpha \neq \tau$ if there exist p', q' such that $p \stackrel{\tau}{\Rightarrow} p' \stackrel{\alpha}{\rightarrow} q' \stackrel{\tau}{\Rightarrow} q$.

The definition of *weak early ground bisimulation* S is then the following:

$$\forall p,q.\ p\,S\,q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{ if } p \stackrel{\alpha}{\rightarrow} p' \text{ with } \alpha \neq x(y) \wedge \mathrm{bn}(\alpha) \cap \mathrm{fn}(q) = \varnothing, \\ \qquad \text{then } \exists q'.\ q \stackrel{\alpha}{\Rightarrow} q' \text{ and } p'\,S\,q' \\ \forall x, y, p'.\ \text{ if } p \stackrel{x(y)}{\longrightarrow} p' \text{ with } y \notin \mathrm{fn}(q), \\ \qquad \text{then } \forall w.\ \exists q'.\ q \stackrel{x(y)}{\Longrightarrow} q' \text{ and } p'[^w/_y]\,S\,q'[^w/_y] \\ \text{(and vice versa)} \end{cases}$$

So we define the corresponding *weak early bisimilarity* $\approx_E$ as follows:

$$p \approx_E q \quad \Leftrightarrow \quad p\, S\, q \text{ for some weak early ground bisimulation } S.$$

It is possible to define *weak late ground bisimulation* and *weak late bisimilarity* $\approx_L$ in a similar way (see Problem 13.9).

As the reader can expect, weak (early and late) bisimilarities are not congruences due to aliasing, as it was already the case for strong bisimilarities. In addition, weak (early and late) bisimilarities are not congruences for a choice context, as it was already the case for CCS. Both problems can be fixed by combining the solutions we have shown for weak observational congruence in CCS and for strong (early and late) full bisimilarities.

Problems

13.1. The *asynchronous* π-calculus allows only outputs with no continuation, i.e., it allows output atoms of the form $\bar{x}\langle y\rangle$ but not output prefixes, yielding a smaller calculus.[5] Show that any process in the original π-calculus can be represented in the asynchronous π-calculus using an extra (fresh) channel to simulate explicit acknowledgement of name transmission. Since a continuation-free output can model a message-in-transit, this fragment shows that the original π-calculus, which is intuitively based on synchronous communication, has an expressive asynchronous communication model inside its syntax.

13.2. The *polyadic* π-calculus allows communicating more than one name in a single action:

$$\bar{x}\langle z_1, \ldots z_n\rangle.P \text{ (polyadic output) and } \quad x(z_1, \ldots z_n).P \quad \text{(polyadic input).}$$

Show that this polyadic extension can be encoded in the monadic calculus (i.e., the ordinary π-caculus) by passing the name of a private channel through which the multiple arguments are then transmitted, one-by-one, in sequence.

13.3. A *higher order* π-calculus can be defined where not only names but processes are sent through channels, i.e., action prefixes of the form $x(Y).p$ and $\bar{x}\langle P\rangle.p$ are allowed where Y is a process variable and P a process. Davide Sangiorgi established the surprising result that the ability to pass processes does not increase the expressivity of the π-calculus: passing a process P can be simulated by just passing a name that points to P instead. Formalise this intuition by showing how to encode higher-order processes in ordinary ones.

13.4. Prove that $x \notin \mathrm{fn}(p)$ implies $(x)p \equiv p$, where $\equiv$ is the structural congruence.

13.5. Exhibit two π-calculus agents p and q such that $p \simeq_E q$ but $\mathrm{fn}(p) \neq \mathrm{fn}(q)$.

[5] Equivalently, one can take the fragment of the π-calculus such that for any subterm of the form $\bar{x}y.p$ it must be $p = \mathbf{nil}$.

13.6. As needed in the proof of the Harmony Lemma 13.1, prove that for any structural equivalence axiom $p \equiv p'$ and for any transition $p' \xrightarrow{\alpha} q'$ then there exists a transition $p \xrightarrow{\alpha} q$ for some $q \equiv q'$.

13.7. Prove the following properties for the π-calculus, where $\sim_E$ is the strong early ground bisimilarity:

$$(x)(p \mid q) \sim_E p \mid (x)q \text{ if } x \notin \mathrm{fn}(p) \quad (x)(p \mid q) \sim_E p \mid (x)q \quad (x)(p \mid q) \sim_E ((x)p) \mid (x)q.$$

offering counterexamples if the properties do not hold.

13.8. Prove that strong early ground bisimilarity is a congruence for the restriction operator. Distinguish the case of input action. Assume that if S is a bisimulation, also $S' = \{(\sigma(x), \sigma(y)) | (x,y) \in S\}$ is a bisimulation, where σ is a one-to-one renaming.

13.9. Spell out the definition of *weak late ground bisimulation* and *weak late bisimilarity* $\approx_L$.

13.10. In the π-calculus, infinite branching is a serious drawback for finite verification. Show that agents

$$p \stackrel{\text{def}}{=} x(y).\bar{y}y.\mathbf{nil} \qquad q \stackrel{\text{def}}{=} (y)\bar{x}y.\bar{y}y\mathbf{nil}$$

are infinitely branching. Modify the input axiom, the open rule, and possibly the parallel composition rule by limiting to one the number of different fresh names which can be assigned to the new name. Modify also the input clause for the early bisimulation by limiting the set of possible continuations by substituting all the free names and only one fresh name. Discuss the possible criteria for choosing the fresh name, e.g., the first, in some order, name which is not free in the agent. Check if your criteria make agents p and r bisimilar or not, where

$$r \stackrel{\text{def}}{=} x(y).(\bar{y}y.\mathbf{nil} \mid (z)\bar{z}w.\mathbf{nil})$$

(note that $(z)\bar{z}w.\mathbf{nil}$ is just a deadlock component).

Part V
Probabilistic Systems

This part focuses on models and logics for probabilistic and stochastic systems. Chapter 14 presents the theory of random processes and Markov chains. Chapter 15 studies (reactive and generative) probabilistic models of computation with observable actions and sources of non-determinism together with a specification logic. Chapter 16 defines the syntax, operational and abstract semantics of PEPA, a well-known high-level language for the specification and analysis of stochastic, interactive systems.

Chapter 14
Measure Theory and Markov Chains

The future is independent of the past, given the present. (Markov property as folklore)

Abstract Future is largely unpredictable. Non-determinism accounts for modelling some phenomena arising in reactive systems, but it does not allow a quantitative estimation of how likely is one event w.r.t. another. We use the term *random* or *probability* to denote systems where the quantitative estimation is possible. In this chapter we present well-studied models of probabilistic systems, called *random processes* and *Markov chains* in particular. The second come in two flavours, depending on the underlying model of time (discrete or continuous). Their key feature is called *Markov property* and it allows to develop an elegant theoretical setting, where it can be conveniently estimated, e.g., how long a system will sojourn in a given state, or the probability of finding the system in a given state at a given time or in the long run. We conclude the chapter by discussing how bisimilarity equivalences can be extended to Markov chains.

14.1 Probabilistic and Stochastic Systems

In previous chapters we have exploited non-determinism to represent choices and parallelism. Probability can be viewed as a refinement of non-determinism, where it can be expressed that some choices are more likely or more frequent than others. We distinguish two main cases: *probabilistic* and *stochastic* models.

Probabilistic models associate a probability to each operation. If many operations are enabled at the same time, then the system uses the probability measure to choose the action that will be executed next. As we will see in Chapter 15, models with many different combinations of probability, non-determinism and observable actions have been studied.

In *stochastic* models each event has a duration. The model binds a random variable to each operation. This variable represents the time necessary to execute the operation. The models we will study use exponentially distributed variables, associating a rate to each event. Often in stochastic systems there is no explicit non-deterministic choice: when a race between events is enabled, the fastest operation is actually chosen.

We start this chapter by introducing some basic concepts of measure theory on which we will rely in order to construct probabilistic and stochastic models. Then we will present one of the most used stochastic models, called *Markov chains*. A Markov chain, named after the Russian mathematician Andrey Markov (1856–1922), is characterised by the the fact that the probability to evolve from one state to another depends only on the current state and not on the sequence of events that preceded it (e.g., it does not depend on the states traversed before reaching the current one). This feature, called the *Markov property*, essentially states that the system is memoryless, or rather that the relevant information about the past is entirely contained in the present state. A Markov chain allows to predict important statistical properties about the future behaviour of a system. We will discuss both the discrete time and the continuous time variants of Markov chains and we will examine some interesting properties which can be studied relying on probability theory.

14.2 Probability Space

A probability space accounts for modelling experiments with some degree of randomness. It comprises a set Ω of all possible outcomes (called *elementary events*) and a set $\mathscr{A}$ of *events* that we are interested in. An event is just a set of outcomes, i.e., $\mathscr{A} \subseteq \wp(\Omega)$, but in general we are not interested in the whole powerset $\wp(\Omega)$, especially because when Ω is infinite, then we would not be able to assign reasonable probabilities to all events in $\wp(\Omega)$. However, the set $\mathscr{A}$ should include at least the impossible event $\varnothing$ and the certain event Ω. Moreover, since events are sets, it is convenient to require that $\mathscr{A}$ is closed under the usual set operations. Thus if A and B are events, then also their intersection $A \cap B$, their union $A \cup B$ and complement $\overline{A}$ should be event, so that we can express, e.g., probabilities about the fact that two events will happen together, or about the fact that some event is not going to happen. If this is the case, then $\mathscr{A}$ is called a *field*. We call it a σ-field if it is also closed under countable union of events. A σ-field is indeed the starting point to define measurable spaces and hence probability spaces.

Definition 14.1 (σ-field). Let Ω be a set of elementary events and $\mathscr{A} \subseteq \wp(\Omega)$ be a family of subsets of Ω, then $\mathscr{A}$ is a σ-field if all of the following hold:

1. $\varnothing \in \mathscr{A}$ (the impossible event is in $\mathscr{A}$);
2. $\forall A \in \mathscr{A} \Rightarrow (\Omega \setminus A) \in \mathscr{A}$ ($\mathscr{A}$ is closed under complement);
3. $\forall \{A_n\}_{n \in \mathbb{N}} \subseteq \mathscr{A}. \bigcup_{i \in \mathbb{N}} A_i \in \mathscr{A}$ ($\mathscr{A}$ is closed under countable union).

The elements of $\mathscr{A}$ are called *events*.

Remark 14.1. It is immediate to see that $\mathscr{A}$ must include the certain event (i.e., $\Omega \in \mathscr{A}$, by 1 and 2) and that also the intersection of a countable sequence of elements of $\mathscr{A}$ is in $\mathscr{A}$, i.e., $\bigcap_{i \in \mathbb{N}} A_i = \Omega \setminus (\bigcup_{i \in \mathbb{N}} (\Omega \setminus A_i))$ (it follows by 2, 3 and the De Morgan property).

Let us illustrate the notion of σ-field by showing a simple example over a finite set of events.

Example 14.1. Let $\Omega = \{a,b,c,d\}$, we define a σ-field on Ω by setting $\mathscr{A} \subseteq \wp(\Omega)$:

$$\mathscr{A} = \{\varnothing, \{a,b\}, \{c,d\}, \{a,b,c,d\}\}$$

The smallest σ-field associated with a set Ω is $\{\varnothing, \Omega\}$ and the smallest σ-field that includes an event A is $\{\varnothing, A, \Omega \setminus A, \Omega\}$. More generally, given any subset $\mathscr{B} \subseteq \wp(\Omega)$ there is a least σ-field that contains $\mathscr{B}$.

σ-fields fix the domain on which we define a particular class of functions called *measures*, which assign a real number to each measurable set of the space. Roughly, a measure can be seen as a notion of size that we wish to attach to sets.

Definition 14.2 (Measure). Let $(\Omega, \mathscr{A})$ be a σ-field. A function $\mu : \mathscr{A} \to [0, +\infty]$ is a *measure* on $(\Omega, \mathscr{A})$ if all of the following hold:

1. $\mu(\varnothing) = 0$;
2. for any countable collection $\{A_n\}_{n \in \mathbb{N}} \subseteq \mathscr{A}$ of pairwise disjoint sets we have $\mu(\bigcup_{i \in \mathbb{N}} A_i) = \sum_{i \in \mathbb{N}} \mu(A_i)$.

A set contained in $\mathscr{A}$ is then called a *measurable set*, and the pair $(\Omega, \mathscr{A})$ is called *measurable space*. We are interested to a particular class of measures called *probabilities*. A probability is a essentially a "normalised" measure.

Definition 14.3 (Probability). A measure P on $(\Omega, \mathscr{A})$ is a *probability* if $P(\Omega) = 1$.

It is immediate from the definition of probability that the codomain of P cannot be the whole set $\mathbb{R}$ of real numbers but it is just the interval of reals $[0,1]$.

Definition 14.4 (Probability space). Let $(\Omega, \mathscr{A})$ be a measurable space and P be a probability on $(\Omega, \mathscr{A})$, then $(\Omega, \mathscr{A}, P)$ is called a *probability space*.

14.2.1 Constructing a σ-field

Obviously one can think that in order to construct a σ-field that contains some sets equipped with a probability it is enough to construct the closure of these sets (together with top and bottom elements) under complement and countable union. But it comes out from set theory that not all sets are measurable. More precisely, it has been shown that it is not possible to define (in ZFC set theory) a probability for all the subsets of Ω when its cardinality is[1] $2^{\aleph_0}$ (i.e., there is no function $P : \wp(\mathbb{R}) \to [0,1]$ that satisfies Definition 14.4). So we have to be careful in defining a σ-field on a set Ω of elementary events that is uncountable.

The next example shows how this problem can be solved in a special case.

[1] The symbol $\aleph_0$, called *aleph zero*, is the smallest infinite cardinal, i.e., it denotes the cardinality of $\mathbb{N}$. Thus $2^{\aleph_0}$ is the cardinality of the powerset $\wp(\mathbb{N})$ as well as of the continuum $\mathbb{R}$.

Example 14.2 (Coin tosses). Let us consider the classic coin toss experiment. We have a fair coin and we want to model sequences of coin tosses. We would like to define Ω as the set of infinite sequences of head (H) and tail (T):

$$\Omega = \{H,T\}^{\infty}.$$

Unfortunately this set has cardinality $2^{\aleph_0}$. As we have just said a measure on uncountable sets does not exist. So we can restrict our attention to a countable set: the set $\mathscr{C}$ of finite sequences of coin tosses. In order to define a σ-field which can account for almost all the events that we could express in words, we define the following set for each $\alpha \in \mathscr{C}$ called the *shadow* of α:

$$[\alpha] = \{\ \omega \in \Omega \mid \exists \omega' \in \Omega.\ \alpha\omega' = \omega\ \}$$

The shadow of α is the set of infinite sequences of which α is a prefix. The right hand side of Figure 14.1 shows graphically the set $[\alpha]$ of infinite paths corresponding to the finite sequence α.

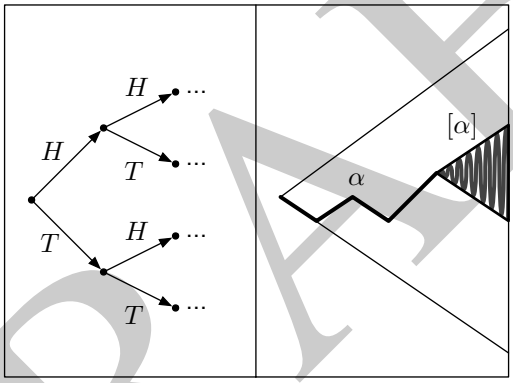


Fig. 14.1: The shadow of α

Now the σ-field which we were looking for is the one generated by the shadows of the sequences in $\mathscr{C}$. In this way we can start by defining a probability measure P on the σ-field generated by the shadows of $\mathscr{C}$, then we can assign a non-zero probability to (all finite sequences and) some infinite sequences of coin tosses by setting:

$$p(\omega) = \begin{cases} P([\omega]) & \text{if } \omega \text{ is finite} \\ P\left(\bigcap_{\alpha\in\mathscr{C},\ \omega\in[\alpha]} [\alpha]\right) & \text{if } \omega \text{ is infinite} \end{cases}$$

For the second case, remind that the definition of σ-field ensures that countable intersection of measurable sets is measurable. Measure theory results show that this measure exists and is unique.

Very often we have structures that are associated with a topology (e.g. there exists a standard topology, called Scott topology, associated to each CPO) so it is useful to define a standard method to obtain a σ-field from a topology.

Definition 14.5 (Topology). Let T be a set and $\mathscr{T} \subseteq \wp(T)$ be a family of subsets of T. Then $\mathscr{T}$ is said to be a *topology* on T if:

- $T, \varnothing \in \mathscr{T}$;
- $A, B \in \mathscr{T} \Rightarrow A \cap B \in \mathscr{T}$, i.e., the topology is closed under finite intersection;
- let $\{A_i\}_{i \in I}$ be any family of sets in $\mathscr{T}$ then $\bigcup_{i \in I} A_i \in \mathscr{T}$, i.e., the topology is closed under finite and infinite union.

The pair $(T, \mathscr{T})$ is said to be a *topological space*.

We call A an *open* set if it is in $\mathscr{T}$ and it is a *closed* set if $T \setminus A$ is open.

Remark 14.2. Note that in general a set can be open, closed, both or neither. For example, T and $\varnothing$ are open and also closed sets. Open sets should not be confused with measurable sets, because measurable sets are closed under complement and countable intersection. This difference makes the notion of measurable function very different from that of continuous function.

Definition 14.6 (Borel σ-field). Let $\mathscr{T}$ be a topology, we call the *Borel* σ-field of $\mathscr{T}$ the smallest σ-field that contains $\mathscr{T}$.

It turns out that the σ-field generated by the shadows which we have seen in the previous example is the Borel σ-field generated by the topology associated with the CPO of sets of infinite paths ordered by inclusion.

Example 14.3 (Euclidean topology). The *euclidean topology* is a topology on real numbers whose open sets are open intervals of real numbers:

$$]a,b[\quad = \quad \{x \in \mathbb{R} \mid a < x < b\}$$

We can extend the topology to the correspondent Borel σ-field, then associating to each open interval its length we obtain the usual *Lebesgue* measure.

It is often convenient to work with a generating collection, because Borel σ-fields are difficult to describe directly.

14.3 Continuous Random Variables

Stochastic processes associate a(n exponentially distributed) *random variable* to each event in order to represent its timing. So the concept of random variable and distribution will be central to the development in this chapter.

Suppose that an experiment has been performed and its outcome $\omega \in \Omega$ is known. A (continuous) random variable associates a real number to ω, e.g., by observing

some of its features. For example, if ω is a finite sequence of coin tosses, a random variable X can count how many heads appear in ω. Then we can try to associate a probability measure on the possible values of X. However, it turns out that in general we cannot define a function $f : \mathbb{R} \to [0,1]$ such that $f(x)$ is the probability that X is x, because the set $\{\omega \mid X(\omega) = x\}$ is not necessarily an element of a measurable space. We consider instead (measurable) sets of the form $\{\omega \mid X(\omega) \leq x\}$.

Definition 14.7 (Random variable). Let $(\Omega, \mathscr{A}, P)$ be a probability space, a function $X : \Omega \to \mathbb{R}$ is said to be a *random variable* if

$$\forall x \in \mathbb{R}.\ \{\omega \in \Omega \mid X(\omega) \leq x\} \in \mathscr{A}.$$

The condition expresses the fact that for each real number x, we can assign a probability to the set $\{\omega \in \Omega \mid X(\omega) \leq x\}$, because it is included in a measurable space. Notice that if we take as $(\Omega, \mathscr{A})$ the measurable space of the real numbers with the Lebesgue measure, the identity $id : \mathbb{R} \to \mathbb{R}$ satisfies the above condition. As another example, we can take sequences of coin tosses, assign the digit 0 to head and 1 to tail and see the sequences as binary representations of decimals in $[0,1)$.

Random variables can be classified by considering the set of their values. We call *discrete* a random variable that has a numerable or finite set of possible values. We say that a random variable is *continuous* if the set of its values is continuous. In the remainder of this section we will consider mainly continuous variables.

A random variable is completely characterised by its *probability law* which describes the probability that the variable will be found in a value less than or equal to the parameter.

Definition 14.8 (Cumulative distribution function). Let $S = (\Omega, \mathscr{A}, P)$ be a probability space, $X : \Omega \to \mathbb{R}$ be a continuous random variable over S. We call *cumulative distribution function* (also *probability law*) of X the image of P through X and denote it by $F_X : \mathbb{R} \to [0,1]$, i.e.:

$$F_X(x) \stackrel{\text{def}}{=} P(\{\omega \in \Omega \mid X(\omega) \leq x\}).$$

Note that the definition of random variable guarantees that, for any $x \in \mathbb{R}$, the set $\{\omega \in \Omega \mid X(\omega) \leq x\}$ is assigned a probability. Moreover, if $x < y$ then $F_X(x) \leq F_X(y)$.

As a matter of notation, we write $P(X \leq a)$ to mean $F_X(a)$, from which we derive:

$$P(X > a) \stackrel{\text{def}}{=} P(\{\omega \in \Omega \mid X(\omega) > a\}) = 1 - F_X(a)$$
$$P(a < X \leq b) \stackrel{\text{def}}{=} P(\{\omega \in \Omega \mid a < X(\omega) \leq b\}) = F_X(b) - F_X(a).$$

The other important function which describes the relative probability of a continuous random variable to take a specified value is the *probability density*.

Definition 14.9 (Probability density). Let $X : \Omega \to \mathbb{R}$ be a continuous random variable on the probability space $(\Omega, \mathscr{A}, P)$. We call the integrable function $f_X : \mathbb{R} \to [0, \infty)$ the *probability density* of X if:

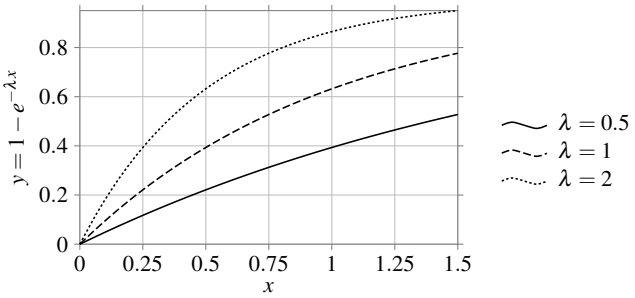


Fig. 14.2: Exponential probability laws with different rates λ

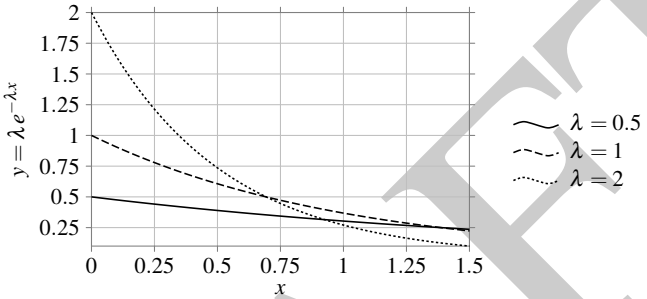


Fig. 14.3: Exponential density distributions with different rates λ

$$\forall a,b \in \mathbb{R}.\ P(a < X \leq b) = \int_a^b f_X(x)dx$$

So we can define the probability law F_X of a variable X with density f_X as follows:

$$F_X(a) = \int_{-\infty}^{a} f_X(x)dx$$

Note that $P(X = a) \stackrel{\text{def}}{=} P(\{\omega \mid X(\omega) = a\})$ is usually 0 when continuous random variables are considered. In case X is a discrete random variable, then its distribution function has jump discontinuities and the function $f_X : \mathbb{R} \to [0,1]$ given by $f_X(x) \stackrel{\text{def}}{=} P(X = x)$ is called *probability mass function*.

We are particularly interested in exponentially distributed random variables.

Definition 14.10 (Exponential distribution). A continuous random variable X is said to be *exponentially distributed* with parameter λ if its probability law and density function are defined as follows:

$$F_X(x) = \begin{cases} 1 - e^{-\lambda x} & \text{if } x \geqslant 0 \\ 0 & x < 0 \end{cases} \qquad f_X(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geqslant 0 \\ 0 & x < 0 \end{cases}$$

The parameter λ is called the *rate* of X and it characterises the expected value (mean) of X, which is $1/\lambda$, and the variance of X, which is $1/\lambda^2$. Some plottings of the functions F_X and f_X associated with exponential distributions with different rates are illustrated in Figure 14.2 and 14.3.

One of the most important features of exponentially distributed random variables is that they are memoryless, meaning that the current value of the random variable does not depend on the previous values.

Example 14.4 (Radioactive Atom). Let us consider a radioactive atom, which due to its instability can easily loose energy. It turns out that the probability that an atom will decay is constant over the time. So this system can be modelled by using an exponentially distributed, continuous random variable whose rate is the decay rate of the atom. Since the random variable is memoryless we have that the probability that the atom will decay at time t_0+t knowing that it is not decaying yet at time t_0 is the same for any choice of t_0, as it depends just on t.

In the following we denote by $P(A \mid B)$ the conditional probability of the event A given the event B, with

$$P(A \mid B) \stackrel{\text{def}}{=} \frac{P(A \cap B)}{P(B)}.$$

Theorem 14.1 (Memoryless). *Let X be an exponentially distributed (continuous) random variable with rate λ. Then:*

$$P(X \leq t_0 + t \mid X > t_0) = P(X \leq t).$$

Proof. Since X is exponentially distributed, its probability law is:

$$F_X(t) = \int_0^t \lambda e^{-\lambda x} dx$$

so we need to prove:

$$\frac{P(t_0 < X \leq t_0 + t)}{P(X > t_0)} = \frac{\int_{t_0}^{t_0+t} \lambda e^{-\lambda x} dx}{\int_{t_0}^{\infty} \lambda e^{-\lambda x} dx} \stackrel{?}{=} \int_0^t \lambda e^{-\lambda x} dx = P(X \leq t)$$

Since $\int_a^b \lambda e^{-\lambda x} dx = \left[-e^{-\lambda x}\right]_a^b = \left[e^{-\lambda x}\right]_b^a$ it follows that:

$$\frac{\int_{t_0}^{t_0+t} \lambda e^{-\lambda x} dx}{\int_{t_0}^{\infty} \lambda e^{-\lambda x} dx} = \frac{\left[e^{-\lambda x}\right]_{t_0+t}^{t_0}}{\left[e^{-\lambda x}\right]_{\infty}^{t_0}} = \frac{e^{-\lambda t_0} - e^{-\lambda t} \cdot e^{-\lambda t_0}}{e^{-\lambda t_0}} = \frac{\cancel{e^{-\lambda t_0}}(1 - e^{-\lambda t})}{\cancel{e^{-\lambda t_0}}} = 1 - e^{-\lambda t}$$

We conclude by:

$$\int_0^t \lambda e^{-\lambda x} dx = \left[e^{-\lambda x}\right]_t^0 = 1 - e^{-\lambda t}.$$

□

Another interesting feature of exponentially distributed random variables is the easy way in which we can compose information in order to find the probability of more complex events. For example if we have two random variables X_1 and X_2 which represent the delay of two events e_1 and e_2, we can try to calculate the probability that either of the two events will be executed before a specified time t. As we will see it happens that we can define an exponentially distributed random variable whose cumulative probability is the probability that either e_1 or e_2 executes before a specified time t.

Theorem 14.2. *Let X_1 and X_2 be two exponentially distributed continuous random variables with rates respectively λ_1 and λ_2 then:*

$$P(\min\{X_1, X_2\} \leq t) = 1 - e^{-(\lambda_1+\lambda_2)t}$$

Proof. We recall that for any two events (not necessarily disjoint) we have:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

and that for two independent events we have

$$P(A \cap B) = P(A) \times P(B).$$

Then:

$$\begin{aligned} P(\min\{X_1, X_2\} \leq t) &= P(X_1 \leq t \vee X_2 \leq t) \\ &= P(X_1 \leq t) + P(X_2 \leq t) - P(X_1 \leq t \wedge X_2 \leq t) \\ &= P(X_1 \leq t) + P(X_2 \leq t) - P(X_1 \leq t) \times P(X_2 \leq t) \\ &= (1 - e^{-\lambda_1 t}) + (1 - e^{-\lambda_2 t}) - (1 - e^{-\lambda_1 t})(1 - e^{-\lambda_2 t}) \\ &= 1 - e^{-\lambda_1 t} e^{-\lambda_2 t} \\ &= 1 - e^{-(\lambda_1+\lambda_2)t} \end{aligned}$$

□

Thus $X = \min\{X_1, X_2\}$ is also an exponentially distributed random variable, whose rate is $\lambda_1 + \lambda_2$. We will exploit this property to define, e.g., the sojourn time in continuous time Markov chains (see Section 14.4.4).

A second important value that we can calculate is the probability that an event will be executed before another. This corresponds in our view to calculate the probability that X_1 will take a value smaller than the one taken by X_2, namely that the action associated with X_1 is chosen instead of the one associated with X_2.

Theorem 14.3. *Let X_1 and X_2 be two exponentially distributed, continuous random variables with rate respectively λ_1 and λ_2 then:*

$$P(X_1 < X_2) = \frac{\lambda_1}{\lambda_1 + \lambda_2}$$

Proof. Imagine you are at some time t and neither of the two variables has fired. The probability that X_1 fires in the infinitesimal interval dt while X_2 fires in any successive instant is

$$\lambda_1 e^{-\lambda_1 t}\left(\int_t^\infty \lambda_2 e^{-\lambda_2 t_2} dt_2\right) dt$$

From which we derive:

$$\begin{aligned}
P(X_1 < X_2) &= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1}\left(\int_{t_1}^\infty \lambda_2 e^{-\lambda_2 t_2} dt_2\right) dt_1 \\
&= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1}\left[e^{-\lambda_2 t_2}\right]_\infty^{t_1} dt_1 \\
&= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1}\cdot e^{-\lambda_2 t_1} dt_1 \\
&= \int_0^\infty \lambda_1 e^{-(\lambda_1+\lambda_2) t_1} dt_1 \\
&= \left[\frac{\lambda_1}{\lambda_1+\lambda_2} e^{-(\lambda_1+\lambda_2)t}\right]_\infty^0 \\
&= \frac{\lambda_1}{\lambda_1+\lambda_2}.
\end{aligned}$$

□

We will exploit this property when presenting the process algebra PEPA, in Chapter 16.

As a special case, when the rates of the two variables are equal, i.e., $\lambda_1 = \lambda_2$, then $P(X_1 < X_2) = 1/2$.

14.3.1 Stochastic Processes

Stochastic processes are a very powerful mathematical tool that allows us to describe and analyse a wide variety of systems.

Definition 14.11 (Stochastic process). Let $(\Omega, \mathscr{A}, P)$ be a probability space and T be a set, then a family $\{X_t\}_{t\in T}$ of random variables over Ω is said to be a *stochastic process*.

A stochastic process can be identified with a function $X : \Omega \times T \to \mathbb{R}$ such that:

$$\forall t \in T.\ X(\cdot, t) : \Omega \to \mathbb{R} \text{ is a random variable.}$$

Usually the values in $\mathbb{R}$ that each random variable can take are called *states* and the element of T are interpreted as times.

Obviously the set T strongly characterises the process. A process in which T is $\mathbb{N}$ or a subset of $\mathbb{N}$ is said to be a *discrete time* process; on the other hand if $T = \mathbb{R}$

(or $T = [0,\infty)$) then the process is a *continuous time* process. The same distinction is usually done on the value that each random variable can assume: if this set has a countable or finite cardinality then the process is *discrete*; otherwise it is *continuous*. We will focus only on discrete processes with both discrete and continuous time. When the set $S = \{x \mid \exists \omega \in \Omega, t \in T.X(\omega,t) = x\}$ of states is finite, with cardinality N, without loss of generality, we can assume that $S = \{1,2,...,N\}$ is just the set of the first N positive natural numbers and we read $X_t = i$ as "the stochastic process X is in the ith state at time t".

14.4 Markov Chains

Stochastic processes studied by classical probability theory often involve only independent variables, namely the outcomes of the process are totally independent from the past. *Markov chains* extend the classic theory by dealing with processes where each variable is influenced by the previous one. This means that in Markov processes the next outcome of the system is influenced only by the previous state. One could think to extend this theory in order to allow general dependencies between variables, but it turns out that it is very difficult to prove general results on processes with dependent variables. We are interested in Markov chains since they provide an expressive mathematical framework to represent and analyse important interleaving and sequential systems.

Definition 14.12 (Markov chain). Let $(\Omega, \mathscr{A}, P)$ be a probability space, T be a totally ordered set and $\{X_t\}_{t\in T}$ be a stochastic process. Then, $\{X_t\}_{t\in T}$ is said to be a *Markov chain* if for each sequence $t_0 < ... < t_n < t_{n+1}$ of times in T and for all states $x, x_0, x_1, ..., x_n \in \mathbb{R}$:

$$P(X_{t_{n+1}} = x \mid X_{t_n} = x_n, \ldots, X_{t_0} = x_0) = P(X_{t_{n+1}} = x \mid X_{t_n} = x_n).$$

The previous proposition is usually referred to as *Markov property*.

An important characteristic of a Markov chain is the way in which it is influenced by the time. We have two types of Markov chains, *inhomogeneous* and *homogeneous*. In the first case the state of the system depends on the time, namely the probability distribution changes over time. In homogeneous chains on the other hand the time does not influence the distribution, i.e., the transition probability does not change during the time. We will consider only the simpler case of homogeneous Markov chains, gaining the possibility to shift the time axis back and forward.

Definition 14.13 (Homogeneous Markov chain). Let $\{X_t\}_{t\in T}$ be a Markov chain; it is *homogeneous* if for all states $x, x' \in \mathbb{R}$ and for all times $t, t' \in T$ with $t < t'$ we have:

$$P(X_{t'} = x' | X_t = x) = P(X_{t'-t} = x' | X_0 = x).$$

In what follows we use the term "Markov chain" as a synonym for "homogeneous Markov chain".

14.4.1 Discrete and Continuous Time Markov Chain

As we said, one of the most important things about stochastic processes in general, and about Markov chains in particular, is the choice of the set of times. In this section we will introduce two kinds of Markov chains, those in which $T = \mathbb{N}$, called *discrete time Markov chain* (DTMC), and those in which $T = \mathbb{R}$, referred to as *continuous time Markov chain*.

Definition 14.14 (Discrete time Markov Chain (DTMC)). Let $\{X_t\}_{t\in\mathbb{N}}$ be a stochastic process; then, it is a *discrete time Markov chain* (DTMC) if for all $n \in \mathbb{N}$ and for all states $x, x_0, x_1, ..., x_n \in \mathbb{R}$:

$$P(X_{n+1} = x \mid X_n = x_n, \ldots, X_0 = x_0) = P(X_{n+1} = \mid X_n = x_n).$$

Since we are restricting our attention to homogeneous chains then we can reformulate the Markov property as follows:

$$P(X_{n+1} = x \mid X_n = x_n, \ldots, X_0 = x_0) = P(X_1 = x \mid X_0 = x_n)$$

Assuming the possible states are $1, ..., N$, the DTMC is entirely determined by the transition probabilities $a_{i,j} = P(X_1 = j \mid X_0 = i)$ for $i, j \in \{1, ..., N\}$.

Definition 14.15 (Continuous time Markov Chain (CTMC)). Let $\{X_t\}_{t\in\mathbb{R}}$ be a stochastic process; then, it is a *continuous time Markov chain* (CTMC) if for all states $x, x_0, ..., x_n$, for any $\Delta_t \in [0, \infty)$ and any sequence of times $t_0 < ... < t_n$ we have:

$$P(X_{t_n+\Delta_t} = x \mid X_{t_n} = x_n, \ldots, X_{t_0} = x_0) = P(X_{t_n+\Delta_t} = x \mid X_{t_n} = x_n).$$

As for the discrete case, the homogeneity allows to reformulate the Markov property as follows:

$$P(X_{t_n+\Delta_t} = x \mid X_{t_n} = x_n, \ldots, X_{t_0} = x_0) = P(X_{\Delta_t} = x \mid X_0 = x_n).$$

Assuming the possible states are $1, ..., N$, the CTMC is entirely determined by the rates $\lambda_{i,j}$ that govern the probability $P(X_t = j \mid X_0 = i) = 1 - e^{-\lambda_{i,j}t}$.

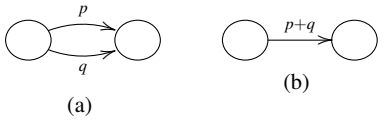
We remark that the exponential random variable is the only continuous random variable with the memoryless property, i.e., CTMC are necessarily exponentially distributed.

14.4.2 DTMC as LTS

A DTMC can be viewed as a particular LTS whose labels are probabilities. Usually such LTS are called *probabilistic transition systems* (PTS).

A difference between LTS and PTS is that in LTS we can have structures like the one shown in Figure 14.4(a), with two transitions that are co-initial and co-final

and carry different labels. In PTS we cannot have this kind of situation since two different transitions between the same pair of states have the same meaning of a single transition labeled with the sum of the probabilities, as shown in Figure 14.4(b).



(a) (b)

Fig. 14.4: Two equivalent DTMCs

The PTS (S, α_D) associated with a DTMC has a set of states S and a transition function $\alpha_D : S \to (\mathscr{D}(S) \cup 1)$ where $\mathscr{D}(S)$ denotes the set of discrete probability distributions over S and $1 = \{*\}$ is a singleton used to represent the deadlock states. We recall that a discrete probability distribution over a set S is a function $D : S \to [0,1]$ such that $\sum_{s \in S} D(s) = 1$.

Definition 14.16 (PTS of a DTMC). Let $\{X_t\}_{t \in \mathbb{N}}$ be a DTMC whose set of states is S. Its corresponding PTS has set of states S and transition function $\alpha_D : S \to (\mathscr{D}(S) \cup 1)$ defined as follows:

$$\alpha_D(s) = \begin{cases} \lambda s'.\ P(X_1 = s' \mid X_0 = s) & \text{if } s \text{ is not a deadlock state} \\ * & \text{otherwise.} \end{cases}$$

Note that for each non-deadlock state s it holds:

$$\sum_{s' \in S} \alpha_D(s)(s') = 1.$$

Usually the transition function is represented through a matrix P whose indices i, j represent states s_i, s_j and each element $a_{i,j}$ is the probability that knowing that the system is in the state i it would be in the state j in the next time instant, namely $\forall i, j \leq |S|.\ a_{i,j} = \alpha_D(s_i)(s_j)$, note that in this case each row of P must sum to one. This representation allows us to study the system by relaying on linear algebra. In fact we can represent the present state of the system by using a row vector $\pi^{(t)} = [\pi_i^{(t)}]_{i \in S}$ where $\pi_i^{(t)}$ represents the probability that the system is in the state s_i at the time t. If we want to calculate how the system will evolve (i.e., the next state distribution) starting from this state we can simply multiply the vector with the matrix which represents the transition function, as the following example of a three state system shows:

$$\pi^{(t+1)} = \pi^{(t)} P = \begin{vmatrix} \pi_1^{(t)} & \pi_2^{(t)} & \pi_3^{(t)} \end{vmatrix} \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = \begin{vmatrix} a_{1,1}\pi_1^{(t)} + a_{2,1}\pi_2^{(t)} + a_{3,1}\pi_3^{(t)} \\ a_{1,2}\pi_1^{(t)} + a_{2,2}\pi_2^{(t)} + a_{3,2}\pi_3^{(t)} \\ a_{1,3}\pi_1^{(t)} + a_{2,3}\pi_2^{(t)} + a_{3,3}\pi_3^{(t)} \end{vmatrix}^{\mathsf{T}}$$

where the resulting row vector is transposed for space matter.

For some special class of DTMCs we can prove the existence of a limit vector for $t \to \infty$, that is to say the probability that the system is found in a particular state is stationary in the long run (see Section 14.4.3).

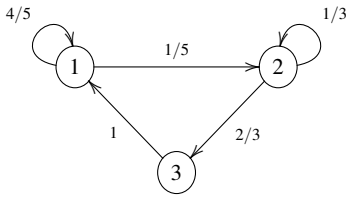


Fig. 14.5: A DTMC

Example 14.5 (DTMC). Let us consider the DTMC in Figure 14.5. We represent the chain algebraically by using the following matrix:

$$P = \begin{vmatrix} 4/5 & 1/5 & 0 \\ 0 & 1/3 & 2/3 \\ 1 & 0 & 0 \end{vmatrix}$$

Now suppose that we do not know the state of the system at time t, thus we assume the system has equal probability $\frac{1}{3}$ of being in any of the three states. We represent this situation with the following vector:

$$\pi^{(t)} = \begin{vmatrix} 1/3 & 1/3 & 1/3 \end{vmatrix}$$

Now we can calculate the state distribution at time $t+1$ as follows:

$$\pi^{(t+1)} = \begin{vmatrix} 1/3 & 1/3 & 1/3 \end{vmatrix} \begin{vmatrix} 4/5 & 1/5 & 0 \\ 0 & 1/3 & 2/3 \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 3/5 & 8/45 & 2/9 \end{vmatrix}$$

Notice that the sum of probabilities in the result $3/5+8/45+2/9$ is again 1. Obviously we can iterate this process in order to simulate the evolution of the system.

Since we have represented a Markov chain by using a transition system it is quite natural to ask for the probability of a finite path.

Definition 14.17 (Finite path probability). Let $\{X_t\}_{t\in\mathbb{N}}$ be a DTMC and $s_1 \cdots s_n$ a finite path of its PTS (i.e., $\forall i.\ 1 \leq i < n \ \Rightarrow\ \alpha_D(s_i)(s_{i+1}) > 0$) we define the probability $P(s_1 \cdots s_n)$ of the path $s_1 \cdots s_n$ as follows:

$$P(s_1 \cdots s_n) = \prod_{i=1}^{n-1} \alpha_D(s_i)(s_{i+1}) = \prod_{i=1}^{n-1} a_{s_i,s_{i+1}}.$$

Example 14.6 (Finite paths). Let us consider the DTMC of Example 14.5 and take the path 1 2 3 1. We have:

$$P(1\ 2\ 3\ 1) = a_{1,2} \times a_{2,3} \times a_{3,1} = \frac{1}{5} \times \frac{2}{3} \times 1 = \frac{2}{15}$$

Note that if we consider the sequence of states 1 1 3 1:

$$P(1\ 1\ 3\ 1) = a_{1,1} \times a_{1,3} \times a_{3,1} = \frac{4}{5} \times 0 \times 1 = 0$$

In fact there is no transition allowed from state 1 to 3.

Note that it would make no sense to define the probability of infinite paths as the product of the probabilities of all choices, because any infinite sequence would have a null probability. We can overcome this problem by using the Borel σ-field generated by the shadows, as seen in Example 14.2.

14.4.3 DTMC Steady State Distribution

In this section we will present a special class of DTMCs which guarantees that the probability that the system is found in a state can be estimated on the long term. This means that the probability distribution of each state of the DTMC (i.e., the corresponding value in the vector $\pi^{(t)}$) reaches a *steady state distribution* which does not change in the future, namely if π_i is the steady state distribution for the state i, if $\pi_i^{(0)} = \pi_i$ then $\pi_i^{(t)} = \pi_i$ for each $t > 0$.

Definition 14.18 (Steady state distribution). We define the *steady state distribution* (or *stationary distribution*) $\pi = \left|\pi_1 \dots \pi_n\right|$ of a DTMC as the limit distribution:

$$\forall i \in [1,n].\ \pi_i = \lim_{t\to\infty} \pi_i^{(t)}$$

when such limit exists.

In order to guarantee that the limit exists we will restrict our attention to a subclass of Markov chains.

Definition 14.19 (Ergodic Markov chain). Let $\{X_t\}_{t\in\mathbb{N}}$ be a Markov chain then it is said to be *ergodic* if it is both:

irreducible: each state is reachable from each other; and
aperiodic: the gcd[2] of the lengths of all paths from any state to itself must be 1.

Theorem 14.4. *Let $\{X_t\}_{t\in\mathbb{N}}$ be an ergodic Markov chain. Then the steady state probability π always exists and it is independent from the initial state probability distribution.*

[2] The gcd is the greatest common divisor.

The steady state probability distribution π can be computed by solving the system of linear equations:

$$\pi = \pi\, P$$

where P is the matrix associated to the chain, under the additional constraint that the sum of all probabilities is 1.

Example 14.7 (Steady state distribution). Let us consider the DTMC of Example 14.5. It is immediate to check that it is ergodic. To find the steady state distribution we need to solve the following linear system:

$$\begin{vmatrix}\pi_1 & \pi_2 & \pi_3\end{vmatrix} \begin{vmatrix} 4/5 & 1/5 & 0 \\ 0 & 1/3 & 2/3 \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix}\pi_1 & \pi_2 & \pi_3\end{vmatrix}$$

The corresponding system of linear equations is

$$\begin{cases} \frac{4}{5}\pi_1 + \pi_3 = \pi_1 \\ \frac{1}{5}\pi_1 + \frac{1}{3}\pi_2 = \pi_2 \\ \frac{2}{3}\pi_2 = \pi_3 \end{cases}$$

Note that the equations express the fact that the probability to be in the state i is given by the sum of the probabilities to be in any other state j weighted by the probability to move from j to i. By solving the system of linear equations we obtain the solution:

$$\begin{vmatrix} 10\pi_2/3 & \pi_2 & 2\pi_2/3 \end{vmatrix}$$

i.e., $\pi_1 = \frac{10}{3}\pi_2$ and $\pi_3 = \frac{2}{3}\pi_2$.

Now by imposing $\pi_1 + \pi_2 + \pi_3 = 1$ we have $\pi_2 = 1/5$ thus:

$$\pi = \begin{vmatrix} 2/3 & 1/5 & 2/15 \end{vmatrix}$$

So, independently from the initial state, in the long run it is more likely to find the system in the state 1 than in states 2 or 3, because the steady state probability of being in state 1 is much larger than the other two probabilities.

14.4.4 CTMC as LTS

Also continuous time Markov chains can be represented as LTSs, but in this case the labels are rates and not probabilities. We have two equivalent definitions for the transition function:

$$\alpha_C : S \to S \to \mathbb{R} \qquad \text{or} \qquad \alpha_C : (S \times S) \to \mathbb{R}$$

where S is the set of states of the chain and any real value $\lambda = \alpha_C(s)(s')$ (or $\lambda = \alpha_C(s_1, s_2)$) represents the rate which labels the transition $s \xrightarrow{\lambda} s'$. Also in this case, likewise DTMC, we have that two different transitions between the same two states are merged in a single transition whose label is the sum of the rates. We write $\lambda_{i,j}$ for the rate $\alpha_C(s_i, s_j)$ associated with the transition from state s_i to state s_j. A difference here is that the self loops can be ignored: this is due to the fact that in continuous time we allow the system to *sojourn* in a state for a period and staying in a state is indistinguishable from moving to the same state via a loop.

The probability that some transition happens from state s_i in some time t can be computed by taking the minimum of the continuous random variables associated with the possible transitions: by Theorem 14.2 we know that such probability is also exponentially distributed and has a rate that is given by the sum of rates of all the transitions outgoing from s_i.

Definition 14.20 (Sojourn time). Let $\{X_t\}$ a CTMC. The probability that no transition happens from a state s_i in some (sojourn) time t is 1 minus the probability that some transition happens:

$$\forall t \in (0, \infty).\ P(X_t = s_i \mid X_0 = s_i) = e^{-\lambda t} \text{ with } \lambda = \sum_{j \neq i} \lambda_{i,j}.$$

As for DTMCs we can represent a CTMC by using linear algebra. In this case the matrix Q which represents the system is defined by setting $q_{i,j} = \alpha_C(s_i, s_j) = \lambda_{i,j}$ when $i \neq j$ and $q_{i,i} = -\sum_{j \neq i} q_{i,j}$. This matrix is usually called *infinitesimal generator*. This definition is convenient for steady state analysis, as explained by the end of the next section.

14.4.5 Embedded DTMC of a CTMC

Often the study of a CTMC results very hard particularly in term of computational complexity. So it is useful to have a standard way to discretise the CTMC by synthesising a DTMC, called *embedded DTMC*, in order to simplify the analysis.

Definition 14.21 (Embedded DTMC). Let α_C be the transition function of a CTMC. Its *embedded DTMC* has the same set of states S and transition function α_D defined by taking:

$$\alpha_D(s_i)(s_j) = \begin{cases} \frac{\alpha_C(s_i, s_j)}{\sum_{s \neq s_i} \alpha_C(s_i, s)} & \text{if } s_i \neq s_j \\ 0 & \text{otherwise.} \end{cases}$$

As we can see, the previous definition simply normalises to 1 the rates in order to calculate a probability.

While the embedded DTMC completely determines the probabilistic behaviour of the system, it does not fully capture the behaviour of the continuous time process because it does not specify the rates at which transitions occur.

Regarding the steady state analysis, since in the infinitesimal generator matrix Q describing the CTMC we have $q_{i,i} = -\sum_{j \neq i} q_{i,j}$ for any state index i, the steady state distribution can equivalently be computed by solving the system of (homogeneous, normalised) linear equations $\pi\, Q = 0$ (see Problem 14.11).

14.4.6 CTMC Bisimilarity

Obviously, since Markov chains can be seen as a particular type of LTS, one could think to modify the notion of bisimilarity in order to study the equivalence between stochastic systems.

Let us start by revisiting the notion of LTS bisimilarity in a slightly different way from that seen in Chapter 11.

Definition 14.22 (Reachability predicate). Given and LTS $(S, L, \rightarrow)$, we define a function $\gamma : S \times L \times \wp(S) \rightarrow \{true, false\}$ which takes a state p, an action ℓ and a set of states I and returns *true* if there exists a state $q \in I$ reachable from p with a transition labelled by ℓ, and *false* otherwise. Formally, given an equivalence class of states I we define:

$$\gamma(p, \ell, I) \stackrel{\text{def}}{=} \exists q \in I.\ p \xrightarrow{\ell} q.$$

Suppose we are given a (strong) bisimulation relation R. We know that its induced equivalence relation $\equiv_R$ is also a bisimulation. Let I be an equivalence classes induced by R. By definition of bisimulation we have that taken any two states $s_1, s_2 \in I$ if $s_1 \xrightarrow{\ell} s_1'$ for some ℓ and s_1' then it must be the case that there exists s_2' such that $s_2 \xrightarrow{\ell} s_2'$ and s_2' is in the same equivalence class I' as s_1' (and vice versa).

Now consider the function $\Phi : \wp(S) \rightarrow \wp(S)$ defined by letting:

$$\forall p, q \in S.\ p\ \Phi(R)\ q \stackrel{\text{def}}{=} \big(\forall \ell \in L.\ \forall I \in S_{/\equiv_R}.\ \gamma(p, \ell, I) \Leftrightarrow \gamma(q, \ell, I) \big)$$

where I ranges over the equivalence classes induced by the relation R.

Definition 14.23 (Bisimulation revisited). By the argument above, a (strong) bisimulation is just a relation such that $R \subseteq \Phi(R)$ and the largest bisimulation is the bisimilarity relation defined as

$$\simeq \stackrel{\text{def}}{=} \bigcup_{R \subseteq \Phi(R)} R.$$

The construction Φ can be extended to the case of CTMCs. The idea is that equivalent states will fall into the same equivalence class and that if a state has multiple transitions with rates $\lambda_1, \ldots, \lambda_n$ to different states $s_1, \ldots, s_n$ that are in the same equivalence class, then we can represent all such transitions by a single transitions that carries the rate $\sum_{i=1}^{n} \lambda_i$. To this aim, given a CTMC $\alpha_C : (S \times S) \rightarrow \mathbb{R}$, we define

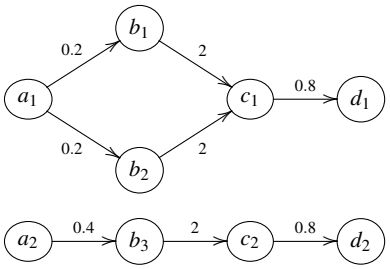


Fig. 14.6: CTMC bisimilarity

a function $\gamma_C : S \times \wp(S) \to \mathbb{R}$ simply by extending the transition function to sets of states as follows:

$$\gamma_C(s,I) = \sum_{s' \in I} \alpha_C(s,s')$$

As we have done above for LTSs, we define the function $\Phi_C : \wp(S) \to \wp(S)$ by:

$$\forall s_1, s_2 \in S.\ s_1\ \Phi_C(R)\ s_2 \stackrel{\text{def}}{=} \forall I \in S_{/\equiv_R}.\ \gamma_C(s_1,I) = \gamma_C(s_2,I)$$

meaning that the total rate of reaching any equivalence class of R from s_1 is the same as that of s_2.

Definition 14.24 (CTMC bisimilarity). A *CTMC bisimulation* is a relation R such that $R \subseteq \Phi_C(R)$ and the *CTMC bisimilarity* $\simeq_C$ is the relation

$$\simeq_C \stackrel{\text{def}}{=} \bigcup_{R \subseteq \Phi_C(R)} R.$$

Let us show how this construction works with an example. Abusing the notation, in the following we write $\alpha_C(s,I)$ instead of $\gamma_C(s,I)$.

Example 14.8. Let us consider the two CTMCs in Figure 14.6. We argue that the following equivalence relation R identifies bisimilar states:

$$R = \{\ \{a_1,a_2\}, \{b_1,b_2,b_3\}, \{c_1,c_2\}, \{d_1,d_2\}\ \}.$$

Let us show that R is a CTMC bisimulation: whenever two states are related, we must check that the sum of the rates from them to the states on any equivalence class coincide. For a_1 and a_2, we have

$$\begin{aligned}
\alpha_C(a_1,\{a_1,a_2\}) = \alpha_C(a_2,\{a_1,a_2\}) &= 0\\
\alpha_C(a_1,\{b_1,b_2,b_3\}) = \alpha_C(a_2,\{b_1,b_2,b_3\}) &= 0.4\\
\alpha_C(a_1,\{c_1,c_2\}) = \alpha_C(a_2,\{c_1,c_2\}) &= 0\\
\alpha_C(a_1,\{d_1,d_2\}) = \alpha_C(a_2,\{d_1,d_2\}) &= 0.
\end{aligned}$$

For b_1, b_2, b_3 we have

$$\alpha_C(b_1,\{c_1,c_2\}) = \alpha_C(b_2,\{c_1,c_2\}) = \alpha_C(b_3,\{c_1,c_2\}) = 2.$$

Note that we no longer mention all remaining trivial cases concerned with the other equivalence classes, where α_C returns 0, because there are no transitions to consider. Finally, we have one last non trivial case to check:

$$\alpha_C(c_1,\{d_1,d_2\}) = \alpha_C(c_2,\{d_1,d_2\}) = 0.8.$$

14.4.7 DTMC Bisimilarity

One could think that the same argument about bisimilarity that we have exploited for CTMCs can be also extended to DTMCs. It is easy to show that if a DTMC has no deadlock states, in particular if it is ergodic, then bisimilarity becomes trivial (see Problem 14.1). This does not mean that the concept of bisimulation on ergodic DTMCs is useless, in fact these relations (finer than bisimilarity) can be used to factorise the chain (lumping) in order to study particular properties.

If we consider DTMCs with some deadlock states, then bisimilarity can be non trivial. Take a DTMC $\alpha_D : S \to (\mathscr{D}(S) \cup 1)$. Let us define the function $\gamma_D : S \to \wp(S) \to (\mathbb{R} \cup 1)$ as follows:

$$\gamma_D(s)(I) = \begin{cases} * & \text{if } \alpha_D(s) = * \\ \sum_{s' \in I} \alpha_D(s)(s') & \text{otherwise} \end{cases}$$

Correspondingly, we set $\Phi_D : \wp(S) \to \wp(S)$ to be defined as:

$$\forall s_1, s_2 \in S.\ s_1\ \Phi_D(R)\ s_2 \stackrel{\text{def}}{=} \forall I \in S_{/\equiv_R}.\ \gamma_D(s_1)(I) = \gamma_D(s_2)(I).$$

Definition 14.25 (DTMC bisimulation). A *DTMC bisimulation* is a relation R such that $R \subseteq \Phi_D(R)$ and the *DTMC bisimilarity* $\simeq_D$ is the relation

$$\simeq_D \stackrel{\text{def}}{=} \bigcup_{R \subseteq \Phi_D(R)} R.$$

In this case:

1. Any two deadlock states s_1, s_2 are bisimilar, because

$$\forall I \in \wp(S).\ \gamma_D(s_1)(I) = \gamma_D(s_2)(I) = *.$$

2. Any deadlock state s_1 is separated from any non deadlock state s, as

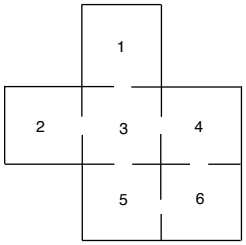
$$\forall I.\ \gamma_D(s_1)(I) = * \neq \gamma_D(s)(I) \in \mathbb{R}.$$

3. If there are no deadlock states, then $\simeq_D = S \times S$.

Problems

14.1. Prove that the bisimilarity relation in a DTMC $\alpha_D : S \to (\mathscr{D}(S) \cup 1)$ without deadlock states (and in particular, when it is ergodic) is always the universal relation $S \times S$.

14.2. A mouse runs through the maze shown below.



At each step it stays in the room or it leaves the room by choosing at random one of the doors (all choices have equal probability).

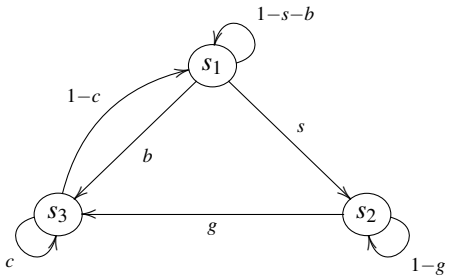
1. Draw the transition graph and give the matrix P for this DTMC.
2. Show that it is ergodic and compute the steady state distribution.
3. Assuming the mouse is initially in room 1, what is the probability that it is in room 6 after three steps?

14.3. Show that the DTMC described by the matrix

$$\begin{vmatrix} \frac{1}{4} & 0 & \frac{3}{4} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

has more than one stationary distribution, actually an infinite number of them. Explain why it is so.

14.4. With the Markov chain below we intend to represent the scenario where Mario, a taxi driver, is looking for costumers. In state s_1, Mario is in the parking place waiting for costumers, which arrive with probability b. Then Mario moves to the busy state s_3, with probabilities c of staying there and $1 - c$ of moving back to s_1. Alternatively, Mario may decide, with probability s, of moving around (state s_2), driving in the busiest streets of town looking for clients, which may show up with probability g.



1. Check that the Markov chain above is ergodic.
2. Compute the steady state probabilities π_1, π_2 and π_3 for the three states s_1, s_2 and s_3 as functions of the parameters b, c, g and s.
3. Evaluate the probabilities for suitable values of the parameters, e.g.,

$$b = 0.5, \quad c = 0.5, \quad g = 0.8, \quad s = 0.3$$

4. Prove that, when it is very likely to find costumers on the streets (i.e., when $g = 1$), in order to maximise π_3, Mario must always move around (i.e., he must choose $s = 1 - b$).

14.5. A state s_i of a Markov chain is called *absorbing* if $\alpha_D(s_i)(s_i) = 1$, and a Markov chain is *absorbing* if it has at least one absorbing state. Can an absorbing Markov chain be ergodic? Explain.

14.6. A machine can be described as being in three different states: (R) under repair, (W) waiting for a new job, (O) operating.

- While the machine is operating the probability to break down is $\frac{1}{20} = 0.05$ and the probability to finish the task (and go to waiting) is $\frac{1}{10} = 0.1$.
- If the machine is under repair there is a $\frac{1}{10} = 0.1$ probability to get repaired, and then the machine will become waiting.
- A broken machine is never brought directly (in one step) to operation.
- If the machine is waiting, there is a $\frac{9}{10} = 0.9$ probability to get into operation.
- A waiting machine does not break.

1. Describe the system as a DTMC, draw the corresponding transition system and define the transition probability matrix. Is it ergodic?
2. Assume that the machine is waiting at time t. What is the probability to be operating at time $t + 1$? Explain.
3. What is the probability that the machine is operating after a long time? Explain.

14.7. A certain calculating machine uses only the digits 0 and 1. It is supposed to transmit one of these digits through several stages. However, at every stage, there is a probability p that the digit that enters this stage will be changed when it leaves and a probability $q = 1 - p$ that it won't.

1. Form a Markov chain to represent the process of transmission. What are the states? What is the matrix of transition probabilities?

2. Assume that the digit 0 enters the machine: what is the probability that the machine, after two stages, produces the digit 0? For which value of p is this probability minimal?

14.8. Consider a CTMC with state space $S = \{0,1\}$. The only possible transitions are described by the rates $q_{0,1} = \lambda$ and $q_{1,0} = \mu$. Compute the following:

1. the embedded DTMC;
2. the state probabilities $\pi^{(t)}$ in terms of the initial distribution $\pi^{(0)}$;

14.9. Consider a CTMC with $N+1$ states representing the number of possible active instances of a service, from 0 to a maximum N. Let i denote the number of currently active instances. A new instance can be spawn with rate

$$\lambda_i \stackrel{\text{def}}{=} (N-i) \times \lambda$$

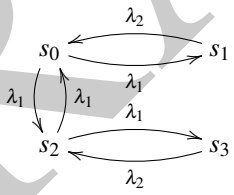
for some fixed λ, i.e., the rate decreases as there are more instances already running,[3] while an instance is terminated with rate

$$\mu_i \stackrel{\text{def}}{=} i \times \mu$$

for some fixed μ, i.e., the rate increases as there are more active instances to be terminated.

1. Model the system as a CTMC;
2. Compute the infinitesimal generator matrix;
3. Find the steady state probability distribution.

14.10. Let us consider the CTMC



1. What is the probability to sojourn in s_0 for some time t?
2. Assume $\lambda_2 > 2\lambda_1$: are there some bisimilar states?

14.11. Prove that computing the steady state distribution of a CTMC by solving the system of (homogeneous, normalised) linear equations $\pi\, Q = 0$ gives the same result as computing the steady state distribution of the embedded DTMC.

[3] Imagine the number of client is fixed, when i instances of the service are already active to serve i clients, then the number of clients that can require a new instance of the service is decreased by i.

Chapter 15
Discrete Time Markov Chains with Actions and Non-determinism

A reasonable probability is the only certainty. (E.W. Howe)

Abstract In this chapter we introduce some advanced probabilistic models that can be defined by enriching the transition functions of PTSs. As we have seen for Markov chains, the transition system representation is very useful since it comes with a notion of bisimilarity. In fact, using the advanced, categorical notion of *coalgebra*, which however we will not develop further, there is a standard method to define bisimilarity just according to the type of the transition function. Also a corresponding notion of Hennessy-Milner logic can be defined accordingly. First we will see two different ways to add observable actions to our probabilistic models, then we will present extensions which combine non-determinism, actions and probabilities.

15.1 Reactive and Generative Models

In this section we show how it is possible to change the transition function of PTSs in order to extend Markov chains with labels that represent actions performed by the system. There are two main cases to consider, called *reactive models* and *generative models*, respectively:

Reactive: In the first case we add actions that are used by the controller to stimulate the system. When we want the system to change its state we give an input action to it which could affect its future state (its reaction). This is the reason why this type of models is called "reactive". Formally, we have that a *reactive probabilistic transition system* (also called *Markov decision process*) is determined by a transition function of the form:[1]

$$\alpha_{\mathsf{r}} : S \to L \to (\mathscr{D}(S) \cup 1)$$

[1] The subscript r stands for "reactive".

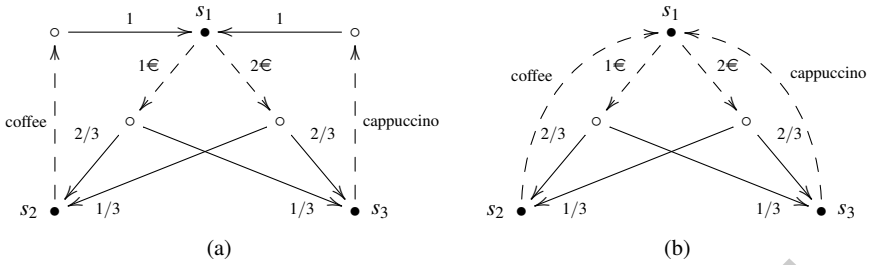


Fig. 15.1: A reactive PTS which represents a coffee maker

where we recall that S is the set of states, $1 = \{*\}$ is a singleton used to represent the deadlock states, and that $\mathscr{D}(S)$ is the set of discrete probability distributions over S.

Generative: In the second case the actions represent the outcomes of the system, this means that whenever the system changes its state it shows an action, whence the terminology "generative". Formally we have that a *generative probabilistic transition system* is determined by a transition function of the form:[2]

$$\alpha_{\mathsf{g}} : S \to (\mathscr{D}(L \times S) \cup 1).$$

Remark 15.1. We have that in a reactive system, for any $s \in S$ and for any $\ell \in L$:

$$\sum_{s' \in S} \alpha_{\mathsf{r}}(s)(\ell)(s') = 1.$$

Instead, in a generative system, for any $s \in S$:

$$\sum_{(\ell,s') \in L \times S} \alpha_{\mathsf{g}}(s)(\ell, s') = 1.$$

This means that in reactive systems, fixed the non-deadlock source state and the action, the next state probabilities must sum to 1, while in a generative system, fixed the non-deadlock source state, the distribution of all transitions must sum to 1 (i.e., given an action ℓ the sum of probabilities to reach any state is less or equal to 1).

15.2 Reactive DTMC

Let us illustrate how a reactive probabilistic system works by using a simple example.

[2] The subscript g stands for "generative".

Example 15.1 (Random coffee maker). Let us consider a system which we call *random coffee maker*, in which the user can insert a coin (1 or 2 euros) then, the coffee maker, based on the value of the input, makes a coffee or a cappuccino with larger or smaller probabilities. The system is represented in Figure 15.1(a). Note that, since we want to allow the system to take input from the environment we have chosen a reactive system to represent the coffee maker. The set of labels is $L = \{1€, 2€, \text{coffee}, \text{cappuccino}\}$ and the corresponding transitions are represented as dashed arrows. There are three states s_1, s_2 and s_3, represented with black-filled circles. If the input $1€$ is received in state s_1, then we can reach state s_2 with probability $2/3$ or s_3 with probability $1/3$, as illustrated by the solid arrows departing from the white-filled circle associated with the distribution. Vice versa, if the input $2€$ is received in state s_1, then we can reach state s_2 with probability $1/3$ or s_3 with probability $2/3$. From state s_2 there is only one transition available, with label coffee, that leads to s_1 with probability 1. Figure 15.1(b) shows a more compact representation of the random coffee maker where the white-filled circle reachable from s_2 is omitted because the probability distribution is trivial. Similarly, from state s_3 there is only one transition available, with label cappucino, that leads to s_1 with probability 1.

As we have shown in the previous chapter, using LTSs we have a standard method to define bisimilarity between probabilistic systems. Take a reactive probabilistic system $\alpha_{\mathsf{r}} : S \to L \to (\mathscr{D}(S) \cup 1)$. Let us define the function $\gamma_{\mathsf{r}} : S \to L \to \wp(S) \to \mathbb{R}$ as follows:

$$\gamma_{\mathsf{r}}(s)(\ell)(I) = \begin{cases} 0 & \text{if } \alpha_{\mathsf{r}}(s)(\ell) = * \\ \sum_{s' \in I} \alpha_{\mathsf{r}}(s)(\ell)(s') & \text{otherwise} \end{cases}$$

Correspondingly, we set $\Phi_{\mathsf{r}} : \wp(S) \to \wp(S)$ to be defined as:

$$\forall s_1, s_2 \in S.\ s_1\ \Phi_{\mathsf{r}}(R)\ s_2 \stackrel{\text{def}}{=} \forall \ell \in L.\ \forall I \in S_{/\equiv_R}.\ \gamma_{\mathsf{r}}(s_1)(\ell)(I) = \gamma_{\mathsf{r}}(s_2)(\ell)(I).$$

Definition 15.1 (Reactive bisimulation). A *reactive bisimulation* is a relation R such that $R \subseteq \Phi_{\mathsf{r}}(R)$ and the *reactive bisimilarity* $\simeq_{\mathsf{r}}$ is the relation

$$\simeq_{\mathsf{r}} \stackrel{\text{def}}{=} \bigcup_{R \subseteq \Phi_{\mathsf{r}}(R)} R.$$

Note that any two bisimilar states s_1 and s_2 must have, for each action, the same probability to reach the states in any other equivalence class.

15.2.1 Larsen-Skou Logic

Now we will present a probabilistic version of Hennessy-Milner logic. This logic has been introduced by Larsen and Skou, and provides a new version of the modal operator. As usual we start from the syntax of Larsen-Skou logic formulas.

Definition 15.2 (Larsen-Skou logic). The formulas of *Larsen-Skou logic* are generated by the following grammar:

$$\varphi \quad ::= \quad \mathit{true} \quad | \quad \varphi_1 \wedge \varphi_2 \quad | \quad \neg\varphi \quad | \quad \langle \ell \rangle_q \varphi.$$

We let $\mathscr{S}$ denote the set of Larsen-Skou logic formulas. The novelty resides in the new modal operator $\langle \ell \rangle_q \varphi$ that takes three parameters: a formula φ, an action ℓ and a real number $q \leq 1$. It corresponds to a refined variant of the usual HM-logic diamond operator $\Diamond_\ell$. Informally, the formula $\langle \ell \rangle_q \varphi$ requires the ability to reach a state satisfying the formula φ by performing the action ℓ with probability at least q.

As we have done for Hennessy-Milner logic we present the Larsen-Skou logic by defining a satisfaction relation $\models \subseteq S \times \mathscr{S}$.

Definition 15.3 (Satisfaction relation). Let $\alpha_r : S \to L \to (\mathscr{D}(S) \cup 1)$ be a reactive probabilistic system. We say that the state $s \in S$ satisfies the Larsen-Skou formula φ and write $s \models \varphi$, if satisfaction can be proved using the (inductively defined) rules:

$$\begin{array}{ll} s \models \mathit{true} & \\ s \models \varphi_1 \wedge \varphi_2 & \text{if } s \models \varphi_1 \text{ and } s \models \varphi_2 \\ s \models \neg\varphi & \text{if } \neg s \models \varphi \\ s \models \langle \ell \rangle_q \varphi & \text{if } \gamma_r(s)(\ell)[\![\varphi]\!] \geq q \text{ where } [\![\varphi]\!] = \{s' \in S \mid s' \models \varphi\}. \end{array}$$

A state s satisfies the formula $\langle \ell \rangle_q \varphi$ if the (sum of the) probability to pass in any state s' that satisfies φ from s with an action labelled ℓ is greater than or equal to q. Note that the corresponding modal operator of the HM-logic can be obtained by setting $q = 1$, i.e., $\langle \ell \rangle_1 \varphi$ means $\Diamond_\ell \varphi$ and we write just $\langle \ell \rangle \varphi$ when this is the case.

Likewise HM-logic, the equivalence induced by Larsen-Skou logic formulas coincide with bisimilarity. Moreover, we have an additional, stronger result: It can be shown that it is enough to consider only the version of the logic without negation.

Theorem 15.1 (Larsen-Skou bisimilarity characterization). *Two states of a reactive probabilistic system are bisimilar if and only if they satisfy the same formulas of Larsen-Skou logic without negation.*

Example 15.2 (Larsen-Skou logic). Let us consider the reactive system in Figure 15.1. We would like to prove that:

$$s_1 \models \langle 1€ \rangle_{1/2} \langle \text{coffee} \rangle \mathit{true}.$$

By definition of the satisfaction relation, we must check that:

$$\gamma_r(s_1)(1\in)(I_1) \geq 1/2 \qquad \text{where} \quad I_1 \stackrel{\text{def}}{=} \{s \in S \mid s \models \langle \text{coffee} \rangle \mathit{true}\}.$$

Now we have that $s \models \langle \text{coffee} \rangle \mathit{true}$ if:

$$\gamma_r(s)(\text{coffee})(I_2) \geq 1 \qquad \text{where} \quad I_2 \stackrel{\text{def}}{=} \{s \in S \mid s \models \mathit{true}\} = \{s_1, s_2, s_3\}.$$

Therefore:

$$I_1 = \{s \mid \gamma_r(s)(\text{coffee})(I_2) \geq 1\} = \{s \mid \gamma_r(s)(\text{coffee})(\{s_1, s_2, s_3\}) \geq 1\} = \{s_2\}.$$

Finally:

$$\gamma_r(s_1)(1\in)\{s_2\} = 2/3 \geq 1/2.$$

15.3 DTMC with Non-determinism

In this section we add non-determinism to generative and reactive systems. Correspondingly, we introduce two classes of models called *Segala automata* and *simple Segala automata*, after the name of Roberto Segala who developed them in 1995. In both cases we use non-determinism to allow the system to choose between different probability distributions.

15.3.1 Segala Automata

Segala automata are generative systems that combine probability and non-determinism. When the system has to move from a state to another, first of all it has to choose non-deterministically a probability distribution, then it uses this information to perform the transition. Formally the transition function of a *Segala automaton* is defined as follows:

$$\alpha_s : S \to \mathscr{P}(\mathscr{D}(L \times S)).$$

As we can see, to each state it corresponds a set of probability distributions $\mathscr{D}(L \times S)$ that are defined on pairs of labels and states. Note that in this case it is not necessary to have the singleton 1 to model explicitly deadlock states, because we can use the empty set to the purpose.

Example 15.3 (Segala automata). Let us consider the system in Figure 15.2. We have an automata with five states, named s_1 to s_5, represented as usual by black-filled circles. When in the state s_1, the system can choose non-deterministically (dashed arrows) between two different distributions d_1 and d_2:

$$\alpha_s(s_1) = \{d_1, d_2\} \quad \text{where} \begin{cases} d_1(\text{flip}, s_2) = 1/2 & d_1(\text{flop}, s_3) = 1/2 \\ d_2(\text{flip}, s_2) = 2/3 & d_2(\text{flop}, s_3) = 1/3 \end{cases}$$

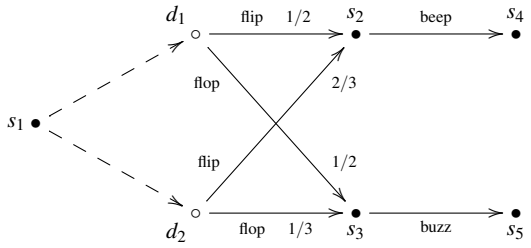


Fig. 15.2: A Segala automata

(we leave implicit that $d_1(l,s) = 0$ and $d_2(l,s) = 0$ for all other label-state pairs).

From states s_2 and s_3 there is just one choice available, respectively the trivial distributions d_3 and d_4 that are omitted from the picture for conciseness of the representation:

$$\alpha_{\mathsf{s}}(s_2) = \{d_3\} \quad \text{where } d_3(\text{beep}, s_4) = 1$$
$$\alpha_{\mathsf{s}}(s_3) = \{d_4\} \quad \text{where } d_4(\text{buzz}, s_5) = 1.$$

Finally, from states s_4 and s_5 there are simply no choices available, i.e., they are deadlock states:

$$\alpha_{\mathsf{s}}(s_4) = \alpha_{\mathsf{s}}(s_5) = \varnothing.$$

15.3.2 Simple Segala Automata

Now we present the reactive version of Segala automata. In this case we have that the system can react to an external stimulation by using a probability distribution. Since we can have more than one distribution for each label, the system uses non-determinism to choose between different distributions for the same label. Formally the transition function of a *simple Segala automaton* is defined as follows:

$$\alpha_{\mathsf{simS}} : S \to \mathscr{P}(L \times \mathscr{D}(S)).$$

Example 15.4 (A Simple Segala Automata). Let us consider the system in Figure 15.3, where we assume some suitable probability value ε has been given. We have six states (represented by black-filled circles, as usual): the state s_1 has two possible inputs, a and c, moreover the label a has associated two different distributions d_1 and d_3, while c has associated a unique distribution d_2. All the other states are deadlock. Formally the system is defined by letting:

$$\alpha_{\mathsf{simS}}(s_1) = \{(a,d_1),(c,d_2),(a,d_3)\} \text{ where } \begin{cases} d_1(s_2) = 1/2 & d_1(s_3) = 1/2 \\ d_2(s_4) = 1/3 & d_2(s_5) = 2/3 \\ d_3(s_1) = \varepsilon & d_3(s_6) = 1-\varepsilon \end{cases}$$

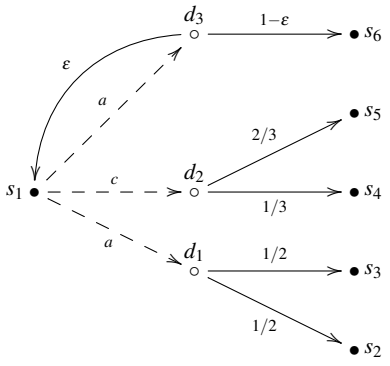


Fig. 15.3: A Simple Segala automaton

and $\alpha_{\mathsf{simS}}(s_2) = \alpha_{\mathsf{simS}}(s_3) = \alpha_{\mathsf{simS}}(s_4) = \alpha_{\mathsf{simS}}(s_5) = \alpha_{\mathsf{simS}}(s_6) = \varnothing$.

15.3.3 Non-determinism, Probability and Actions

As we saw, there are many ways to combine probability, non-determinism and actions. We conclude this chapter by mentioning two other interesting models which can be obtained by redefining the transition function of a PTS.

The first class of systems is that of *alternating transition systems*. In this case we allow the system to perform two types of transition: one using probability distributions and one using non-determinism. An alternating system can be defined formally by a transition function of the form:

$$\alpha_{\mathsf{a}} : S \to (\mathscr{D}(S) + \mathscr{P}(L \times S)).$$

So in this kind of systems we can alternate probabilistic and non-deterministic choices and can partition the states accordingly. (Again, a state s is deadlock when $\alpha_{\mathsf{a}}(s) = \varnothing$).

The second type of systems that we present is that of *bundle transition systems*. In this case the system associates a distribution to subsets of non-deterministic choices. Formally, the transition function has the form:

$$\alpha_{\mathsf{b}} : S \to \mathscr{D}(\mathscr{P}(L \times S)).$$

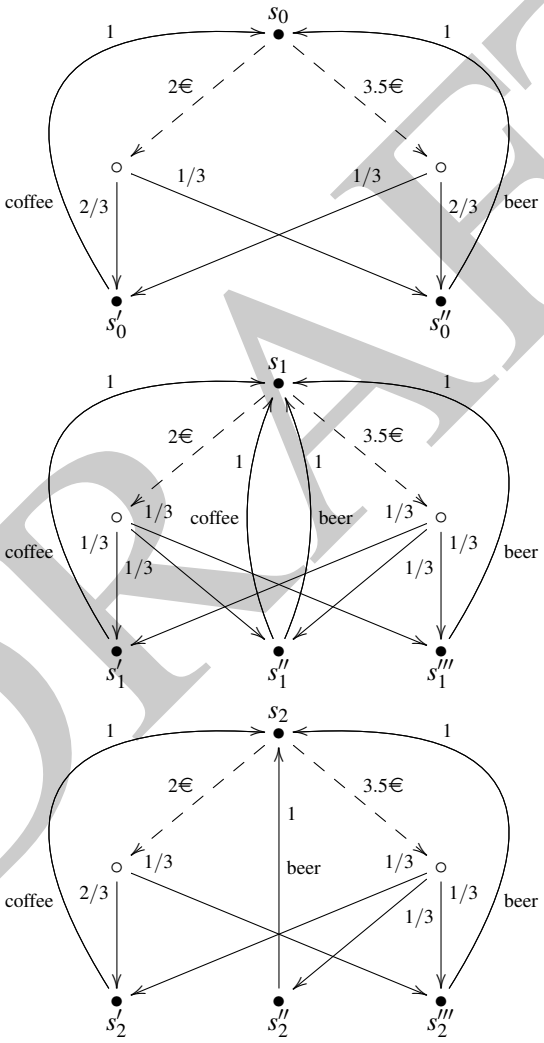
So when a bundle transition system has to perform a transition, first of all it chooses by using a probability distribution a set of possible choices, then non-deterministically it picks one of these.

Problems

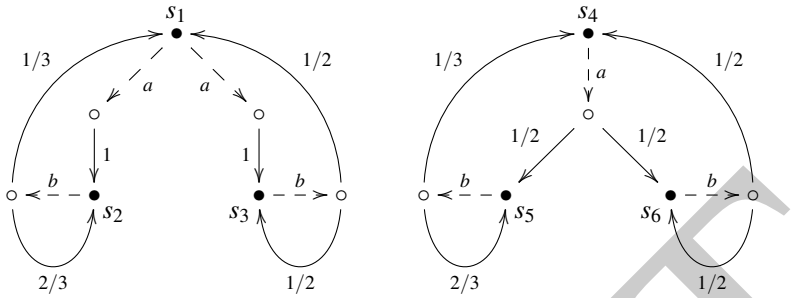
15.1. In which sense is a Segala automaton the most general model?

1. Show in which way a generative LTS, a reactive LTS and a simple Segala automaton can be interpreted as (generative) Segala automata.
2. Explain the difficulties in representing a generative LTS as a simple Segala automaton.

15.2. Consider the following three reactive LTSs. For every pair of systems, check whether their initial states are bisimilar. If they are, describe the bisimulation, if they are not, find a formula of the Larsen-Skou logic that distinguishes them.



15.3. Define formally the notion of bisimulation/bisimilarity for simple Segala automata. Then apply the partition refinement algorithm to the automata below to check which are the bisimilar states.



15.4. Let a *non-stopping*, reactive, probabilistic labelled transition systems (PLTS) be the reactive PLTS with $\alpha : S \to L \to \mathscr{D}(S)$ (rather than $\alpha : S \to L \to (\mathscr{D}(S) \cup 1)$).

1. Prove that all the states of a non-stopping, reactive PLTS are bisimilar.
2. Then give the definition of bisimilarity also for *generative* PLTS.
3. Furthermore, consider the non-stopping subclass of generative PLTS and show an example where some states are not bisimilar.
4. Moreover, give the definition of bisimilarity also for Segala PLTS, and show that Segala bisimilarity reduces to generative PLTS bisimilarity in the deterministic case (namely when, for every state s, $\alpha(s)$ is a singleton).

Chapter 16
PEPA - Performance Evaluation Process Algebra

He who is slowest in making a promise is most faithful in its performance. (Jean Jacques Rousseau)

Abstract The probabilistic and stochastic models we have presented in previous chapters represent system behaviour but not its structure, i.e., they take a monolithic view and do not make explicit how the system is composed and what are the interacting components of which it is made. In this last chapter we introduce a language, called PEPA (Performance Evaluation Process Algebra), for composing stochastic processes and carry out their quantitative analysis. PEPA builds on CSP (Calculus for Sequential Processes), a process algebra similar to CCS but with slightly different primitives. In particular, it relies on multiway communication instead of binary (I/O) one. PEPA actions are labelled with rates and a CTMC can be derived from the LTS of a PEPA process without much efforts to evaluate quantitative properties of the modelled system. The advantage is that the PEPA description of the CTMC remains as a blueprint of the system and allows direct re-use of processes.

16.1 From Qualitative to Quantitative Analysis

To understand the differences between qualitative analysis and quantitative analysis, we remark that qualitative questions like:

- Will the system reach a particular state?
- Does the system implementation match its specification?
- Does a given property ϕ hold within the system?

are replaced by quantitative questions like:

- How long will the system take on average to reach a particular state?
- With what probability does the system implementation match its specification?
- Does a given property ϕ hold within the system within time t with probability p?

Jane Hillston defined the PEPA language in 1994. PEPA has been developed as a high-level language for the description of continuous time Markov chains. Over the years PEPA has been shown to provide an expressive formal language for

modelling distributed systems. PEPA models are obtained as the structured assembly of components that perform individual activities at certain rates and can cooperate on shared actions. The most important features of PEPA w.r.t. other approaches to performance modelling are:

compositionality:	the ability to model a system as the interaction of subsystems, as opposed to poorly modular approaches;
formality:	a rigorous semantics giving a precise meaning to all terms in the language and solving any ambiguities;
abstraction:	the ability to build up complex models from components, disregarding the details when it is appropriate to do so;
separation of concerns:	the ability to model the components and the interaction separately;
structure:	the ability to impose a clear structure to models, which makes them more understandable and easier to maintain;
refinement:	the ability to construct models systematically by refining their specifications;
reusability:	the ability to maintain a library of model components.

For example, queueing networks offer compositionality but not formality; stochastic extensions of Petri nets offer formality but not compositionality; neither offer abstraction mechanisms.

PEPA was obtained by extending CSP (Calculus for Sequential Processes) with probabilities. We start with a brief introduction to CSP, then we will conclude with the presentation of the syntax and operational semantics of PEPA.

16.2 CSP

Communicating Sequential Processes (CSP) is a process algebra introduced by Tony Hoare in 1978 and is a very powerful tool for systems specification and verification. Contrary to CCS, CSP actions have no dual counterpart and the synchronisation between two or more processes is possible when they all perform the same action α (in which case the observable label of the synchronisation is still α). Since during communication the joint action remains visible to the environment, it can be used to interact with other (more than two) processes, realising multiway synchronisation.

16.2.1 Syntax of CSP

We assume that a set Λ of actions α is given. The syntax of CSP processes is defined below, where $L \subseteq \Lambda$ is any set of actions:

$$P, Q \quad ::= \quad \mathbf{nil} \quad | \quad \alpha.P \quad | \quad P + Q \quad | \quad P \underset{L}{\bowtie} Q \quad | \quad P/L \quad | \quad C.$$

We briefly comment on each operator:

nil: is the inactive process;
$\alpha.P$: is a process which can perform an action α and then behaves like P;
$P+Q$: is a process which can choose to behave like P or like Q;
P/L: is the hiding operator; if P performs an action $\alpha \in L$ then P/L performs an unobservable silent action τ;
$P \underset{L}{\bowtie} Q$: is a synchronisation operator, also called *cooperation combinator*. More precisely, it denotes an indexed family of operators, one for each possible set of actions L. The set L is called *cooperation set* and fixes the set of *shared actions* between P and Q. Processes P and Q can use the actions in L to synchronise each other. The actions not included in L are called *individual activities* and can be performed separately by P and Q. As a special case, if $L = \varnothing$ then all the actions of P and Q are just interleaved.
C: is the name, called *constant*, of a recursively defined process that we assume given in a separate set $\Delta = \{C_i \stackrel{\text{def}}{=} P_i\}_{i \in I}$ of declarations.

16.2.2 Operational Semantics of CSP

Now we present the semantics of CSP. As we have done for CCS and π-calculus, we define the operational semantics of CSP as an LTS derived by a set of inference rules. As usual, theorems take the form $P \xrightarrow{\alpha} P'$, meaning that the process P in one step evolves to the process P' by executing the action α.

16.2.2.1 Inactive Process

There is no rule for the inactive process **nil**.

16.2.2.2 Action Prefix and Choice

The rules for action prefix and choice operators are the same as in CCS.

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}.$$

16.2.2.3 Hiding

The hiding operator should not be confused with the restriction operator of CCS: first, hiding takes a set L of labels as a parameter, while restriction takes a single

action; second, when $P \xrightarrow{\alpha} P'$ with $\alpha \in L$ we have that $P/L \xrightarrow{\tau} P'/L$, while $P\backslash\alpha$ blocks the action. Instead, P/L and $P\backslash\alpha$ behaves similarly w.r.t. actions not included in $L \cup \{\alpha\}$.

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin L}{P/L \xrightarrow{\alpha} P'/L} \qquad \frac{P \xrightarrow{\alpha} P' \quad \alpha \in L}{P/L \xrightarrow{\tau} P'/L}.$$

16.2.2.4 Cooperation Combinator

There are three rules for the cooperation combinator $\bowtie_L$: the first two rules allow the interleaving of actions not in L, while the third rule forces the synchronisation of the two processes when performing actions in L. Differently from CCS, when two processes synchronise on α the observed label is still α and not τ.

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{\alpha} P' \underset{L}{\bowtie} Q} \qquad \frac{Q \xrightarrow{\alpha} Q' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{\alpha} P \underset{L}{\bowtie} Q'} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q' \quad \alpha \in L}{P \underset{L}{\bowtie} Q \xrightarrow{\alpha} P' \underset{L}{\bowtie} Q'}.$$

Note that the cooperation combinator is not associative. For example

$$(\alpha.\beta.\mathbf{nil} \underset{\{\alpha\}}{\bowtie} \mathbf{nil}) \underset{\varnothing}{\bowtie} \alpha.\mathbf{nil} \neq (\alpha.\beta.\mathbf{nil}) \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \alpha.\mathbf{nil}).$$

In fact the leftmost process can perform only an action α

$$(\alpha.\beta.\mathbf{nil} \underset{\{\alpha\}}{\bowtie} \mathbf{nil}) \underset{\varnothing}{\bowtie} \alpha.\mathbf{nil} \xrightarrow{\alpha} (\alpha.\beta.\mathbf{nil} \underset{\{\alpha\}}{\bowtie} \mathbf{nil}) \underset{\varnothing}{\bowtie} \mathbf{nil}$$

after which it is deadlock, whereas the rightmost process can perform a synchronisation on α and then it can perform another action β

$$(\alpha.\beta.\mathbf{nil}) \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \alpha.\mathbf{nil}) \xrightarrow{\alpha} (\beta.\mathbf{nil}) \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \mathbf{nil}) \xrightarrow{\beta} \mathbf{nil} \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \mathbf{nil}).$$

16.2.2.5 Constants

Finally, the rule for constants unfolds the recursive definition $C \stackrel{\text{def}}{=} P$, so that C has all transitions that P has.

$$\frac{(C \stackrel{\text{def}}{=} P) \in \Delta \quad P \xrightarrow{\alpha} P'}{C \xrightarrow{\alpha} P'}.$$

16.3 PEPA

As we said, PEPA is obtained by adding probabilities to the execution of actions. As we will see, PEPA processes are stochastic: there are not explicit probabilistic operators in PEPA, but the probabilistic behaviour is obtained by associating an exponentially distributed continuous random variable to each action prefix; this random variable represents the time needed to execute the action. These random variables lead to a clear relationship between the process algebra model and a CTMC. Via this underlying Markov process, performance measures can then be extracted from the model.

16.3.1 Syntax of PEPA

In PEPA an action is a pair (α, r), where α is the action type and r is the rate of the continuous random variable associated with the action. The rate r can be any positive real number. The grammar for PEPA processes is given below:

$$P, Q \quad ::= \quad \textbf{nil} \quad | \quad (\alpha, r).P \quad | \quad P + Q \quad | \quad P \underset{L}{\bowtie} Q \quad | \quad P/L \quad | \quad C$$

$(\alpha, r).P$: is a process which can perform an action α and then behaves like P. In this case the rate r is used to define the exponential variable which describes the duration of the action. A component may have a purely sequential behaviour, repeatedly undertaking one activity after another and possibly returning to its initial state. As a simple example, consider a web server in a distributed system that can serve one request at a time:

$$WS \stackrel{\text{def}}{=} (request, \top).(serve, \mu).(respond, \top).WS.$$

In some cases, as here, the rate of an action falls out of the control of this component: such actions are carried out jointly with another component, with the current component playing some sort of passive role. For example, the web server is passive with respect to the request and respond actions, as it cannot influence the rate at which applications execute these actions. This is recorded by using the distinguished rate $\top$ which we can assume to represent an extremely high value that cannot influence the rates of interacting components.

$P + Q$: has the same meaning of the CSP operator for choice. For example, we can consider an application in a distributed system that can either access a locally available method (with probability p_1) or access to a remote web service (with probability $p_2 = 1 - p_1$). The decision is taken by performing a think action which is parametric to the rate λ:

$$Appl \stackrel{\text{def}}{=} (think, p_1 \cdot \lambda).(local, m).Appl + (think, p_2 \cdot \lambda).(request, rq).(respond, rp).Appl.$$

$P \underset{L}{\bowtie} Q$: has the same meaning of the CSP operator. In the web service example, we can assume that the application and the web server interact over the set of shared actions $L = \{request, respond\}$:

$$Sys \stackrel{\text{def}}{=} (Appl \underset{\varnothing}{\bowtie} Appl) \underset{L}{\bowtie} WS.$$

During the interaction, the resulting action will have the same type of the shared action and a rate reflecting the rate of the slowest action.

P/L: is the same as the CSP hiding operator: the duration of the action is unaffected, but its type becomes τ. In our running example, we may want to hide the local computation of *Appl* to the environment:

$$Appl' \stackrel{\text{def}}{=} Appl/\{local\}.$$

C: is the name of a recursively defined process such as $C \stackrel{\text{def}}{=} P$ that we assume given in a separate set Δ of declarations. Using recursive definitions as the ones given above for *Appl* and *WS*, we are able to describe components with infinite behaviour without introducing an explicit recursion or replication operator.

Usually we are interested only in those agents which have an ergodic underlying Markov process, since we want to apply the steady state analysis. It has been shown that it is possible to ensure ergodicity by using syntactic restrictions on the agents. In particular, the class of PEPA terms which satisfy these syntactic conditions are called *cyclic components* and they can be described by the following grammar:

$$\begin{array}{rcl} P, Q & ::= & S \mid P \underset{L}{\bowtie} Q \mid P/L \\ S, T & ::= & (\alpha, r).S \mid S + T \mid C \end{array}$$

where *sequential processes* S and T can be distinguished from general processes P and Q; and it is required that each recursive process C is sequential, i.e., it must be $(C \stackrel{\text{def}}{=} S) \in \Delta$ for some sequential process S.

16.3.2 Operational Semantics of PEPA

PEPA operational semantics is defined by a rule system similar to the one for CSP. In the case of PEPA, well formed formulas have the form $P \xrightarrow{(\alpha, r)} Q$ for suitable PEPA processes P and Q, activity α and rate r. We assume a set Δ of declarations is available.

16.3.2.1 Inactive Process

As usual, there is no rule for the inactive process **nil**.

16.3.2.2 Action Prefix and Choice

The rules for action prefix and choice are essentially the same as the ones for CSP: the only difference is that the rate r is recorded in the label of transitions.

$$\frac{}{(\alpha,r).P \xrightarrow{(\alpha,r)} P} \qquad \frac{P \xrightarrow{(\alpha,r)} P'}{P+Q \xrightarrow{(\alpha,r)} P'} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q'}{P+Q \xrightarrow{(\alpha,r)} Q'}.$$

16.3.2.3 Constants

The rule for constants is the same as that of CSP, except for the fact transition labels carry also the rate.

$$\frac{(C \stackrel{\text{def}}{=} P) \in \Delta \quad P \xrightarrow{(\alpha,r)} P'}{C \xrightarrow{(\alpha,r)} P'}.$$

16.3.2.4 Hiding

Also the rules for hiding resemble the ones for CSP. Note that when $P \xrightarrow{(\alpha,r)} P'$ with $\alpha \in L$, the rate r is associated with τ in $P/L \xrightarrow{(\tau,r)} P'/L$.

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P/L \xrightarrow{(\alpha,r)} P'/L} \qquad \frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \in L}{P/L \xrightarrow{(\tau,r)} P'/L}.$$

16.3.2.5 Cooperation Combinator

As for CSP, we have three rules for the cooperation combinator. The first two rules are for action interleaving and deserve no further comment.

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P \underset{L}{\bowtie} Q'}.$$

The third rule, called *cooperation* rule (see below), is the most interesting one, because it deals with synchronisation and with the need to combine rates. The cooperation rule exploits the so-called *apparent rate* of action α in P, written $r_\alpha(P)$, which is defined by structural recursion as follows:

$$\begin{aligned}
r_\alpha(\mathbf{nil}) &\stackrel{\text{def}}{=} 0 \\
r_\alpha((\beta,r).P) &\stackrel{\text{def}}{=} \begin{cases} r & \text{if } \alpha = \beta \\ 0 & \text{if } \alpha \neq \beta \end{cases} \\
r_\alpha(P+Q) &\stackrel{\text{def}}{=} r_\alpha(P) + r_\alpha(Q) \\
r_\alpha(P/L) &\stackrel{\text{def}}{=} \begin{cases} r_\alpha(P) & \text{if } \alpha \notin L \\ 0 & \text{if } \alpha \in L \end{cases} \\
r_\alpha(P \underset{L}{\bowtie} Q) &\stackrel{\text{def}}{=} \begin{cases} min(r_\alpha(P), r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q) & \text{if } \alpha \notin L \end{cases} \\
r_\alpha(C) &\stackrel{\text{def}}{=} r_\alpha(P) \quad \text{if } (C \stackrel{\text{def}}{=} P) \in \Delta\ .
\end{aligned}$$

Roughly, the apparent rate $r_\alpha(S)$ is the sum of the rates of all distinct actions α that can be performed by S, thus $r_\alpha(S)$ expresses the overall rate of α in S (because of the property of rates of exponentially distributed variables in Theorem 14.2). Notably, in the case of shared actions, the apparent rate of $P \underset{L}{\bowtie} Q$ is the slowest of the apparent rates of P and Q. The cooperation rule is:

$$\frac{P \xrightarrow{(\alpha,r_1)} P' \quad Q \xrightarrow{(\alpha,r_2)} Q' \quad \alpha \in L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q'} \quad \text{where } r = r_\alpha(P \underset{L}{\bowtie} Q) \times \frac{r_1}{r_\alpha(P)} \times \frac{r_2}{r_\alpha(Q)}$$

Let us now explain the calculation

$$r = r_\alpha(P \underset{L}{\bowtie} Q) \times \frac{r_1}{r_\alpha(P)} \times \frac{r_2}{r_\alpha(Q)}$$

that appears in the cooperation rule. The best way to resolve what should be the rate of the shared action has been a topic of some debate. The definition of cooperation in PEPA is based on the assumption that a component cannot be made to exceed its bounded capacity for carrying out the shared actions, where the bounded capacity consists of the apparent rate of the action. The underlying assumption is that the choice of a specific action (with rate r_i) to carry on the shared activity occurs independently in the two cooperating components P and Q. Now, the probability that a specific action (α, r_i) is chosen by P is (see Theorem 14.3)

$$\frac{r_i}{r_\alpha(P)}.$$

Then, from the choice independence we obtain the combined probability

$$\frac{r_1}{r_\alpha(P)} \times \frac{r_2}{r_\alpha(Q)}.$$

Finally, the resulting rate is the product of the apparent rate

$$r_\alpha(P \underset{L}{\bowtie} Q) = \min(r_\alpha(P), r_\alpha(Q))$$

and the above probability. Notice that if we sum up the rates of all possible synchronisations on α of $P \underset{L}{\bowtie} Q$ we just get $min(r_\alpha(P), r_\alpha(Q))$ (see the example below).

Example 16.1. Let us define two PEPA agents as follows:

$$P \stackrel{\text{def}}{=} (\alpha, r).P_1 + \ldots + (\alpha, r).P_n \qquad Q \stackrel{\text{def}}{=} (\alpha, r).Q_1 + \ldots + (\alpha, r).Q_m$$

for some $n \leq m$. So we have the the following apparent rates:

$$\begin{aligned} r_\alpha(P) &\stackrel{\text{def}}{=} n \times r \\ r_\alpha(Q) &\stackrel{\text{def}}{=} m \times r \\ r_\alpha(P \underset{\{\alpha\}}{\bowtie} Q) &\stackrel{\text{def}}{=} min(r_\alpha(P), r_\alpha(Q)) = n \times r \end{aligned}$$

By the rules for action prefix and choice, we have transitions:

$$P \xrightarrow{(\alpha,r)} P_i \quad \text{for } i \in [1,n] \qquad Q \xrightarrow{(\alpha,r)} Q_j \quad \text{for } j \in [1,m]$$

Then we have $m \times n$ possible ways of synchronising P and Q, :

$$P \underset{\{\alpha\}}{\bowtie} Q \xrightarrow{(\alpha,r')} P_i \underset{\{\alpha\}}{\bowtie} Q_j \quad \text{for } i \in [1,n] \text{ and } j \in [1,m]$$

where

$$r' = (n \times r) \times \frac{r}{n \times r} \times \frac{r}{m \times r} = \frac{r}{m}.$$

So we have $m \times n$ transitions with rate r/m and, in fact, the apparent rate of the synchronisation is:

$$m \times n \times \frac{r}{m} = n \times r = r_\alpha(P \underset{\{\alpha\}}{\bowtie} Q).$$

Remark 16.1. The selection of the exponential distribution as the governing distribution for action durations in PEPA has profound consequences. In terms of the underlying stochastic process, it is the only choice which gives rise to a Markov process. This is due to the memoryless properties of the exponential distribution: the time until the next event is independent of the time since the last event, because the

exponential distribution forgets how long it has already waited. For instance, if we consider the process $(\alpha, r).\,\mathbf{nil} \underset{\varnothing}{\bowtie} (\beta, s).\,\mathbf{nil}$ and the system performs the action α, the time needed to complete β from $\mathbf{nil} \underset{\varnothing}{\bowtie} (\beta, s).\,\mathbf{nil}$ does not need to consider the time already taken to carry out the action α.

The underlying CTMC is obtained from the LTS by associating a (global) state with each process, and the transitions between states are derived from the transitions of the LTS. If in the LTS there are several transitions possible between two processes, since all activity durations are exponentially distributed, in the CTMC there will be a single transition with a total transition rate which is sum of the rates.

The PEPA language is supported by a range of tools and by a wide community of users. PEPA application areas span the subject areas of informatics and engineering. Additional information and a PEPA Eclipse Plug-in are freely available at `http://www.dcs.ed.ac.uk/pepa/`.

We conclude this section by showing a famous example by Jane Hillston of modelling with PEPA.

Example 16.2 (Roland the gunman). We want to model a Far West duel. We have two main characters: Roland the gunman and his enemies. Upon its travels Roland will encounter some enemies with whom he will have no choice but to fight back. For simplicity we assume that Roland has two guns with one bullet in each and that each hit is fatal. We also assume that a sense of honour prevents an enemy from attacking Roland if he is already involved in a gun fight. We model the behaviour of Roland as follows. Normally, Roland is in an idle state $Roland_{\text{idle}}$, but when he is attacked (attacks) he moves to state $Roland_2$, where he has two bullets available in his gun:

$$Roland_{\text{idle}} \stackrel{\text{def}}{=} (\text{attack}, \top).Roland_2.$$

In front of his enemy, Roland can act in three ways: if Roland hits the enemy then he reloads his gun and returns idle; if Roland misses the enemy he tries a second attack (see $Roland_1$); finally if an enemy hits Roland, he dies.

$$\begin{aligned} Roland_2 \stackrel{\text{def}}{=}\ & (\text{hit}, r_{\text{hit}}).(\text{reload}, r_{\text{reload}}).Roland_{\text{idle}} \\ & +(\text{miss}, r_{\text{miss}}).Roland_1 \\ & +(\text{e-hit}, \top).Roland_{\text{dead}}. \end{aligned}$$

The second attempt to shoot by Roland is analogous to the first one, but this time it is the last bullet in Rolands gun and if the enemy is missed no further shot is possible in $Roland_{\text{empty}}$ until the gun is reloaded.

$$\begin{aligned} Roland_1 \stackrel{\text{def}}{=}\ & (\text{hit}, r_{\text{hit}}).(\text{reload}, r_{\text{reload}}).Roland_{\text{idle}} \\ & +(\text{miss}, r_{\text{miss}}).Roland_{\text{empty}} \\ & +(\text{e-hit}, \top).Roland_{\text{dead}}. \\ Roland_{\text{empty}} \stackrel{\text{def}}{=}\ & (\text{reload}, r_{\text{reload}}).Roland_2 \\ & +(\text{e-hit}, \top).Roland_{\text{dead}}. \end{aligned}$$

Finally if Roland is dead he cannot perform any action.

$$Roland_{\text{dead}} \stackrel{\text{def}}{=} \mathbf{nil}.$$

We describe enemies behaviour as follows. If the enemies are idle they can try to attack Roland:

$$Enemies_{\text{idle}} \stackrel{\text{def}}{=} (\text{attack}, r_{\text{attack}}).Enemies_{\text{attack}}.$$

Enemies shoot once and either get hit or they hit Roland.

$$\begin{aligned} Enemies_{\text{attack}} \stackrel{\text{def}}{=} & (\text{e-hit}, r_{\text{e-hit}}).Enemies_{\text{idle}} \\ & +(hit, \top).Enemies_{\text{idle}}. \end{aligned}$$

The rates involved in the model are measured in seconds, so a rate of 1.0 would indicate that the action is expected to occur once every second. We define the following rates:

$\top = $	about ∞
$r_{\text{fire}} = 1$	one shot per second.
$r_{\text{hit-success}} = 0.8$	80% of success.
$r_{\text{hit}} = 0.8$	$r_{\text{fire}} \times r_{\text{hit-success}}$.
$r_{\text{miss}} = 0.2$	$r_{\text{fire}} \times (1 - r_{\text{hit-success}})$.
$r_{\text{reload}} = 0.3$	3 seconds to reload.
$r_{\text{attack}} = 0.01$	Roland is attacked once every 100 seconds.
$r_{\text{e-hit}} = 0.02$	Enemies can hit once every 50 seconds.

So we model the duel as follows:

$$\text{Duel} \stackrel{\text{def}}{=} Roland_{idle} \underset{\{hit, attack, e-hit\}}{\bowtie} Enemies_{\text{idle}}.$$

We can perform various types of analysis of the system by using standard methods. Using the steady state analysis, that we have seen in the previous chapters, we can prove that Roland will always die and the system will deadlock, because there is an infinite supply of enemies (so the system is not ergodic). Moreover we can answer many other questions by using the following techniques:

- Transient analysis: we can ask for the probability that Roland is dead after 1 hour, or the probability that Roland will have killed some enemy within 30 minutes.
- Passage time analysis: we can ask for the probability of passing at least 10 seconds from the first attack to Roland to the time it has hit 3 enemies, or the probability that 1 minute after he is attacked Roland has killed his attacker (i.e., the probability that the model performs a *hit* action within 1 minute after having performed an *attack* action).

Problems

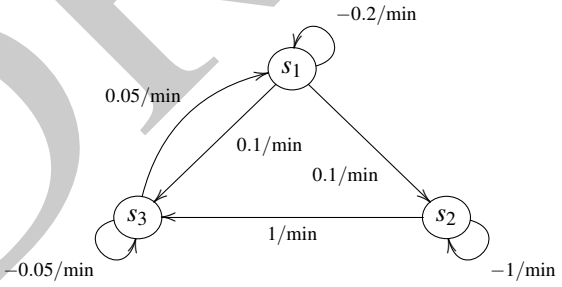
16.1. We have defined CTMC bisimilarity in the case of *unlabeled* transition systems, while PEPA transition system is labeled. Extend the definition of bisimilarity to the labeled version.

16.2. Consider a simple system in which a process repeatedly carries out some task. In order to complete its task the process needs access to a resource for part, but not all, of the time. We want to model the process and the resource as two separate PEPA agents: *Process* and *Resource*, respectively. The *Process* will undertake two activities consecutively: *get* with some rate *rg*, in cooperation with the *Resource*, and *task* at rate *rt*, representing the remainder of its processing task. Similarly the *Resource* will engage in two activities consecutively: *get*, at a rate $rg' > 2rg$ and *update*, at rate *ru*.

1. Give the PEPA specification of a system composed with two *Process*es that compete for one shared *Resource*.
2. What is the apparent rate of action *get* in the initial state of the system?
3. Draw the complete LTS (eight states) of the system and list all its transitions.

16.3. In a multiprocessor system with shared memory, processes must compete to use the memory bus. Consider the case of two identical processes. Each process has cyclic behaviour: it performs some local activity (local action *think*), accesses the bus (synchronization action *get*), operates on the memory (local action *use*) and then releases the bus (synchronization action *rel*). The bus has cyclic behavior with actions *get* and *rel*. Define a PEPA program representing the system and derive the corresponding CTMC (with actions). Find the bisimilar states according to the notion of bisimilarity in Problem 16.1 and draw the minimal CTMC.

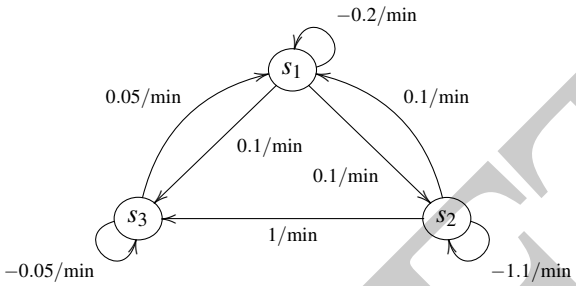
16.4. Consider the taxi driver scenario from Problem 14.3, but this time represented as the CTMC in the Figure below, where rates are defined in 1/minutes, e.g., costumers show up every 10 minutes (rate $0.1/\text{min}$) and rides last 20 minutes (rate $0.05/\text{min}$) .



Assuming a unique label *l* for all the transitions, and disregarding self loops, define a PEPA agent for the system, and show that all states are different in terms of bisimilarity. Finally, to study the steady state behaviour of the system, introduce the

self loops,[1] and write and solve a system of linear equations similar to the one seen for DTMC: $\pi\, Q = 0$ and $\sum_i \pi_i = 1$. The equations express the fact that, for every state s_i, the probability flow from the other states to state s_i is the same as the probability flow from state s_i to the other states.

16.5. Consider the taxi driver scenario from Problem 16.4, but this time with the option of going back to state s_1 (parking) from state s_2 (moving slowly looking for costumers) as in the figure below.



Define a PEPA agent for the system, and show that all states are different in terms of bisimilarity. Finally, to study the steady state behaviour of the system, introduce the self loops, decorated with suitable negative rates, and write and solve a system of linear equations similar to the one seen for DTMC: $\pi\, Q = 0$ and $\sum_i \pi_i = 1$. The equations express the fact that, for every state s_i, the probability flow from the other states to state s_i is the same as the probability flow from state s_i to the other states (see Section 14.4.5).

16.6. Let the (infinitely many) PEPA processes $\{A_\alpha, B_\beta\}$, indexed by strings $\alpha, \beta \in \{0,1\}^*$ be defined as:

$$A_\alpha \stackrel{\text{def}}{=} (a,\lambda).B_{\alpha 0} + (a,\lambda).B_{\alpha 1} \qquad B_\beta \stackrel{\text{def}}{=} (b,\lambda).A_{\beta 0} + (b,\lambda).A_{\beta 1}.$$

Consider the (sequential) PEPA program $P \stackrel{\text{def}}{=} A_\varepsilon$, for ε the empty string:

1. draw (at least in part) the transition system of P;
2. find the states reachable from P;
3. determine the bisimilar states;
4. finally, find the smallest PEPA program bisimilar to P.

16.7. Let the (infinitely many) PEPA processes A_α, indexed by strings $\alpha \in \{0,1\}^*$ be defined as:

$$A_\alpha \stackrel{\text{def}}{=} (a,\lambda).A_{\alpha 0} + (a,\lambda).A_{\alpha 1}.$$

Consider the (sequential) PEPA program $P \stackrel{\text{def}}{=} A_\varepsilon$, for ε the empty string:

[1] Remind that, in the infinitesimal generator matrix of a CTMC, self loops are decorated with negative rates which are negated apparent rates, namely the negated sums of all the outgoing rates.

1. draw (at least in part) the transition system of P;
2. find the states reachable from P;
3. determine the bisimilar states;
4. finally, find the smallest PEPA program bisimilar to P.

16.8. Consider the PEPA process A with

$$A \stackrel{\text{def}}{=} (\alpha,\lambda).B + (\alpha,\lambda).C \qquad B \stackrel{\text{def}}{=} (\alpha,\lambda).A + (\alpha,\lambda).C \qquad C \stackrel{\text{def}}{=} (\alpha,\lambda).A.$$

and derive the corresponding finite state CTMC.

1. What is the probability distribution of staying in B?
2. If $\lambda = 0.1\ sec^{-1}$, what it the probability that the system be still in B after 10 seconds?
3. Are there bisimilar states?
4. Finally, to study the steady state behaviour of the system, introduce the self loops, decorated with suitable negative rates, show that the system is ergodic and write and solve a system of linear equations similar to the one seen for DTMC.

16.9. Consider n *transmitters* $T_0, T_1, \ldots, T_{n-1}$ connected by a token ring. At any moment, a transmitter i can be *ready* to transmit or *not ready*. It becomes ready with a private action *arrive* and a rate λ. Once ready, it stays ready until it transmits, and then it becomes not ready with an action $serve_i$ and rate μ. To resolve conflicts, only the transmitter with the *token* can operate. There is only one token K, which at any moment is located at some transmitter T_i. If transmitter T_i is not ready, the token synchronises with it with an action $walkon_i$ and rate ω moving from transmitter T_i to transmitter $T_{i+1\ (mod\ n)}$. If transmitter T_i is ready, the token synchronises with it with action $serve_i$ and rate μ and stays at transmitter T_i.

Write a PEPA process modelling the above system as follows:

1. define recursively all the states of T_i, for $i \in [0, n-1]$ and of K;
2. define the whole system by choosing the initial state where all transmitters are not ready and the token in at T_0 and composing in parallel all of them with $\underset{L}{\bowtie}$, with L being the set of synchronised actions.
3. Then draw the transition system corresponding to $n = 2$, and compute the bisimilarity relation.
4. Finally define a function f such that $f(n)$ is the number of (reachable) states for the system with n transmitters.

Solutions

Problems of Chapter 2

2.1

1. The strings in L_B are all non-empty sequences of $\mathtt{b}$'s. The strings in L_A are all non-empty sequences of $\mathtt{a}$'s followed by strings in L_B.
2. Letting s^n denote the string obtained by concatenating n replicas of the string s, we have $L_B = \{\mathtt{b}^n \mid n > 0\}$ and $L_A = \{\mathtt{a}^n\mathtt{b}^m \mid n,m > 0\}$.
3. $$\frac{s \in L_A}{\mathtt{a}\ s \in L_A}(1) \qquad \frac{s \in L_B}{\mathtt{a}\ s \in L_A}(2) \qquad \frac{}{\mathtt{b} \in L_B}(3) \qquad \frac{s \in L_B}{\mathtt{b}\ s \in L_B}(4)$$
4. Proof tree:
$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{b} \in L_B}(3)}{\mathtt{b}\,\mathtt{b} \in L_B}(4)}{\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(2)}{\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(1)}{\mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(1)$$
Goal-oriented derivation:
$$\begin{aligned} \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A &\nwarrow \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A \\ &\nwarrow \mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A \\ &\nwarrow \mathtt{b}\,\mathtt{b} \in L_B \\ &\nwarrow \mathtt{b} \in L_B \\ &\nwarrow \square \end{aligned}$$
5. We first prove the correspondence for B, i.e., that $s \in L_B$ is a theorem iff there exists some $n > 0$ with $s = \mathtt{b}^n$. For the 'only if' part, by rule induction, since $s \in L_B$, either $s = \mathtt{b}$ (by rule (3)), or $s = \mathtt{b}\ s'$ for some $s' \in L_B$ (by rule (4)). In the former case, we take $n = 1$ and we are done. In the latter case, by $s' \in L_B$ we have that there is $n' > 0$ with $s' = \mathtt{b}^{n'}$ and take $n = n' + 1$. For the 'if' part, by induction on n, if $n = 1$ we conclude by applying axiom (3); if $n = n' + 1$, we can assume that $\mathtt{b}^{n'} \in L_B$ and conclude by applying rule (4).
Then we prove the correspondence for A, i.e., that $s \in L_A$ is a theorem iff there exists some $n,m > 0$ with $s = \mathtt{a}^n\mathtt{b}^m$. For the 'only if' part, by rule induction, since $s \in L_A$, either $s = \mathtt{a}\ s'$ for some $s' \in L_A$ (by rule (1)), or $s = \mathtt{a}\ s'$ for some

$s' \in L_B$ (by rule (2)). In the former case, by $s' \in L_A$ we have that there is $n', m' > 0$ with $s' = \mathtt{a}^{n'}\mathtt{b}^{n'}$ and take $n = n' + 1$, $m = m'$. In the latter case, by the previous correspondence on B, by $s' \in L_B$ we have that $s' = \mathtt{b}^k$ for some $k > 0$ and conclude by taking $n = 1$ and $m = k$. For the 'if' part, take $s = \mathtt{a}^n\mathtt{b}^m$. By induction on n, if $n = 1$ we conclude by applying axiom (2), since for the previous correspondence we know that $\mathtt{b}^m \in L_B$; if $n = n' + 1$, we can assume that $\mathtt{a}^{n'}\mathtt{b}^m \in L_A$ and conclude by applying rule (1).

2.3

1. The predicate $even(x)$ is a theorem iff x represents an even number (i.e., x is the repeated application of $s(\cdot)$ to 0 for an even number of times).
2. The predicate $odd(x)$ if not a theorem for any x, because there is no axiom.
3. The predicate $leq(x,y)$ is a theorem iff x represents a natural number which is less than or equal to the natural number represented by y.

2.5 Take $t = s(x)$ and $t' = s(y)$.

2.8

$$\begin{aligned} \mathsf{fib}(0,1) &:- . \\ \mathsf{fib}(s(0),1) &:- . \\ \mathsf{fib}(s(s(x)),y) &:- \mathsf{fib}(x,u), \mathsf{fib}(s(x),v), \mathsf{sum}(u,v,y). \end{aligned}$$

2.11 Pgvdrk is intelligent.

Problems of Chapter 3

3.2 Let us denote by c the body of the while command:

$$c \stackrel{\text{def}}{=} \textbf{if } y = 0 \textbf{ then } y := y+1 \textbf{ else skip}$$

Let us take a generic memory σ and consider the goal $\langle w, \sigma\rangle \to \sigma'$.
If $\sigma(y) < 0$ we have:

$$\begin{array}{l} \langle w,\sigma\rangle \to \sigma' \\ \quad \nwarrow_{\sigma'=\sigma} \ \langle y \geq 0, \sigma\rangle \to \textbf{false} \\ \quad\quad \nwarrow^* \ \ \Box \end{array}$$

If instead $\sigma(y) > 0$, we have:

$$\begin{array}{lll} \langle w,\sigma\rangle \to \sigma' & \nwarrow & \langle y \geq 0, \sigma\rangle \to \textbf{true}, \langle c,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\ & \nwarrow^* & \langle c,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\ & \nwarrow^* & \langle \textbf{skip},\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\ & \nwarrow^*_{\sigma''=\sigma} & \langle w,\sigma\rangle \to \sigma' \end{array}$$

Since we reach the same goal from which we started, the command diverges.
Finally, if instead $\sigma(y) = 0$, we have:

$$\begin{array}{lll}
\langle w, \sigma\rangle \to \sigma' & \nwarrow & \langle y \geq 0, \sigma\rangle \to \mathbf{true}\,, \langle c, \sigma\rangle \to \sigma''\,, \langle w, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c, \sigma\rangle \to \sigma''\,, \langle w, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle y := y+1, \sigma\rangle \to \sigma''\,, \langle w, \sigma''\rangle \to \sigma' \\
& \nwarrow^*_{\sigma''=\sigma[^1/_y]} & \langle w, \sigma[^1/_y]\rangle \to \sigma'
\end{array}$$

We reach a goal where w is to be evaluated in a memory $\sigma[^1/_y]$ such that $\sigma[^1/_y](y) > 0$. Thus we are in the previous case and we know that the command diverges.

Summing up, $\langle w, \sigma\rangle \to \sigma'$ iff $\sigma(y) < 0 \wedge \sigma' = \sigma$.

3.4 Let us denote by c' the body of c_2:

$$c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c \textbf{ else skip}$$

We proceed by contradiction. First, assume that there exist σ, σ' such that $\langle c_1, \sigma\rangle \to \sigma'$ and $\langle c_2, \sigma\rangle \not\to \sigma'$. Let us take such σ, σ' for which $\langle c_1, \sigma\rangle \to \sigma'$ has the shortest derivation.

If $\langle b, \sigma\rangle \to \mathbf{false}$, we have

$$\begin{array}{lll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \mathbf{false} \\
& \nwarrow^* & \Box \\
\langle c_2, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \mathbf{false} \\
& \nwarrow^* & \Box
\end{array}$$

Thus it must be $\langle b, \sigma\rangle \to \mathbf{true}$. In this case, we have

$$\begin{array}{lll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \mathbf{true}\,, \langle c, \sigma\rangle \to \sigma''\,, \langle c_1, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c, \sigma\rangle \to \sigma''\,, \langle c_1, \sigma''\rangle \to \sigma' \\
\langle c_2, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \mathbf{true}\,, \langle c', \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c', \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma' \\
& \nwarrow & \langle b, \sigma\rangle \to \mathbf{true}\,, \langle c, \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c, \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma'
\end{array}$$

Now, since σ and σ' were chosen so to allow for the shortest derivation $\langle c_1, \sigma\rangle \to \sigma'$ that cannot be mimicked by $\langle c_2, \sigma\rangle$, it must be the case that $\langle c_1, \sigma''\rangle \to \sigma'$, which is shorter, can still be mimicked, thus $\langle c_2, \sigma''\rangle \to \sigma'$ is provable, but then $\langle c_2, \sigma\rangle \to \sigma'$ holds, leading to a contradiction.

Second, assume that there exist σ, σ' such that $\langle c_2, \sigma\rangle \to \sigma'$ and $\langle c_1, \sigma\rangle \not\to \sigma'$. Then the proof is completed analogously to the previous case.

3.6 Take any σ such that $\sigma(x) = 0$. Then $\langle c_1, \sigma\rangle \to \sigma$, while $\langle c_2, \sigma\rangle \not\to$.

3.9

1. Take $a = 0/0$. Then, for any σ, we have, e.g., $\langle a, \sigma\rangle \to 1$ (since $0 = 0 \times 1$) and $\langle a, \sigma\rangle \to 2$ (since $0 = 0 \times 2$) by straightforward application of rule (div).
2. Take $a = 1/2$. Then, we cannot find an integer n such that $1 = 2 \times n$ and the rule (div) cannot be applied.

Problems of Chapter 4

4.2 We let:

$$\begin{aligned} locs(\textbf{skip}) &\stackrel{\text{def}}{=} \varnothing \\ locs(x := a) &\stackrel{\text{def}}{=} \{x\} \\ locs(c_0; c_1) \;=\; locs(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) &\stackrel{\text{def}}{=} locs(c_0) \cup locs(c_1) \\ locs(\textbf{while } b \textbf{ do } c) &\stackrel{\text{def}}{=} locs(c) \end{aligned}$$

We prove the property

$$P(\langle c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(c).\ \sigma(y) = \sigma'(y)$$

by rule induction.

skip: We need to prove $P(\langle \textbf{skip}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{skip}).\ \sigma(y) = \sigma(y)$ that holds trivially.

assign: We need to prove

$$P(\langle x := a, \sigma\rangle \to \sigma[^n/_x]) \stackrel{\text{def}}{=} \forall y \notin locs(x := a).\ \sigma(y) = \sigma[^n/_x](y)$$

Trivially: $locs(x := a) = \{x\}$ and $\forall y \neq x.\ \sigma[^n/_x](y) = \sigma(y)$.

seq: We assume

$$\begin{aligned} P(\langle c_0, \sigma\rangle \to \sigma'') &\stackrel{\text{def}}{=} \forall y \notin locs(c_0).\ \sigma(y) = \sigma''(y) \\ P(\langle c_1, \sigma''\rangle \to \sigma') &\stackrel{\text{def}}{=} \forall y \notin locs(c_0).\ \sigma''(y) = \sigma'(y) \end{aligned}$$

and we need to prove

$$P(\langle c_0; c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(c_0; c_1).\ \sigma(y) = \sigma'(y)$$

Take $y \notin locs(c_0;c_1) = locs(c_0) \cup locs(c_1)$. It follows that $y \notin locs(c_0)$ and $y \notin locs(c_1)$. By $y \notin locs(c_0)$ and the first inductive hypothesis we have $\sigma(y) = \sigma''(y)$. By $y \notin locs(c_1)$ and the second inductive hypothesis we have $\sigma''(y) = \sigma'(y)$. By transitivity, we conclude $\sigma(y) = \sigma'(y)$.

iftt: We assume

$$P(\langle c_0, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(c_0).\ \sigma(y) = \sigma'(y)$$

and we need to prove

$$P(\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1).\ \sigma(y) = \sigma'(y)$$

Take $y \notin locs(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) = locs(c_0) \cup locs(c_1)$. It follows that $y \notin locs(c_0)$ and hence, by the inductive hypothesis, $\sigma(y) = \sigma'(y)$.

ifff: This case is analogous to the previous one and thus omitted.

whff: We need to prove

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{while } b \textbf{ do } c).\ \sigma(y) = \sigma(y)$$

which is obvious (as for the case of rule skip).

whtt: We assume

$$P(\langle c, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \forall y \notin locs(c).\ \sigma(y) = \sigma''(y)$$
$$P(\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{while } b \textbf{ do } c).\ \sigma''(y) = \sigma'(y)$$

and we need to prove

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{while } b \textbf{ do } c).\ \sigma(y) = \sigma'(y)$$

Take $y \notin locs(\textbf{while } b \textbf{ do } c) = locs(c)$. By the first inductive hypothesis, it follows that $\sigma(y) = \sigma''(y)$, while by the second inductive hypothesis we have $\sigma''(y) = \sigma'(y)$. By transitivity, we conclude $\sigma(y) = \sigma'(y)$.

4.3 We prove the property

$$P(\langle w, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma(x) \geq 0 \ \wedge\ \sigma' = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right]$$

by rule induction. Since the property is concerned with the command w, it is enough to consider the two rules for the while construct.

whff: We assume

$$\langle x \neq 0, \sigma\rangle \to \textbf{false}$$

We need to prove

$$P(\langle w, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \sigma(x) \geq 0 \ \wedge\ \sigma = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right]$$

Since $\langle x \neq 0, \sigma\rangle \to \textbf{false}$ it follows that $\sigma(x) = 0$ and thus $\sigma(x) \geq 0$. Then, $\sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right] = \sigma\left[{}^{0+\sigma(y)}/_{y}, {}^{\sigma(x)}/_{x}\right] = \sigma\left[{}^{\sigma(y)}/_{y}, {}^{\sigma(x)}/_{x}\right] = \sigma$.

whtt: Let $c \stackrel{\text{def}}{=} x := x-1; y := y+1$. We assume

$$\langle x \neq 0, \sigma\rangle \to \textbf{false} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle w, \sigma''\rangle \to \sigma'$$
$$P(\langle w, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma''(x) \geq 0 \ \wedge \ \sigma' = \sigma''\left[{}^{\sigma''(x)+\sigma''(y)}/_{y}, {}^{0}/_{x}\right]$$

We need to prove

$$P(\langle w, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma(x) \geq 0 \ \wedge \ \sigma' = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right]$$

From $\langle c, \sigma\rangle \to \sigma''$ it follows that $\sigma'' = \sigma\left[{}^{\sigma(y)+1}/_{y}, {}^{\sigma(x)-1}/_{x}\right]$. By inductive hypothesis we have $\sigma''(x) \geq 0$, thus $\sigma(x) \geq 1$ and hence $\sigma(x) \geq 0$. Moreover, by inductive hypothesis, we have also

$$\sigma' = \sigma''\left[{}^{\sigma''(x)+\sigma''(y)}/_{y}, {}^{0}/_{x}\right] = \sigma''\left[{}^{\sigma(x)-1+\sigma(y)+1}/_{y}, {}^{0}/_{x}\right] = \sigma''\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right] = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right].$$

4.4 We prove the two implication separately. First we prove the property

$$P(x\, R^{+}\, y) \stackrel{\text{def}}{=} \exists k > 0.\ \exists z_0, \ldots, z_k.\ x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = y$$

by rule induction.

For the first rule

$$\frac{x\, R\, y}{x\, R^{+}\, y}$$

we assume $x\, R\, y$ and we need to prove $P(x\, R^{+}\, y)$. We take $k = 1$, $z_0 = x$ and $z_1 = y$ and we are done.

For the second rule

$$\frac{x\, R^{+}\, y \quad y\, R^{+}\, z}{x\, R^{+}\, z}$$

we assume

$$P(x\, R^{+}\, y) \stackrel{\text{def}}{=} \exists n > 0.\ \exists u_0, \ldots, u_n.\ x = u_0 \ \wedge\ u_0\, R\, u_1 \ \wedge\ \ldots \wedge\ u_{n-1}\, R\, u_n \ \wedge\ u_n = y$$
$$P(y\, R^{+}\, z) \stackrel{\text{def}}{=} \exists m > 0.\ \exists v_0, \ldots, v_m.\ y = v_0 \ \wedge\ v_0\, R\, v_1 \ \wedge\ \ldots \wedge\ v_{m-1}\, R\, v_m \ \wedge\ v_m = z$$

and we need to prove

$$P(x\, R^{+}\, z) \stackrel{\text{def}}{=} \exists k > 0.\ \exists z_0, \ldots, z_k.\ x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = z$$

Take $n, u_0, ..., u_n$ and $m, v_0, ..., v_m$ as provided by the inductive hypotheses. We set $k = n+m$, from which it follows $k > 0$ since $n > 0$ and $m > 0$. Note that $u_n = y = v_0$.

Finally, we let

$$z_i \stackrel{\text{def}}{=} \begin{cases} u_i & \text{if } i \in [0,n] \\ v_{i-n} & \text{if } i \in [n+1,k] \end{cases}$$

and it is immediate to check that the conditions are satisfied.

To prove the reverse implication, we exploit the logical equivalence

$$(\exists k.\, A(k)) \Rightarrow B \quad \Leftrightarrow \quad \forall k.\, (A(k) \Rightarrow B)$$

that holds whenever k does not appear (free) in the predicate B, to prove the universally quantified statement

$$\forall k > 0. \forall x, y. \left((\exists z_0, \ldots, z_k.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = y) \Rightarrow x\, R^+\, y \right)$$

by mathematical induction on k.

The base case is when $k = 1$. Take generic x and y. We assume the premise

$$\exists z_0, z_1.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ z_1 = y$$

and the thesis $x\, R^+\, y$ follows by applying the first inference rule.

For the inductive case, we assume that

$$\forall x, y.\ \left((\exists z_0, \ldots, z_k.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = y) \Rightarrow x\, R^+\, y \right)$$

and we want to prove that

$$\forall x, z.\ \left((\exists z_0, \ldots, z_{k+1}.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_k\, R\, z_{k+1} \ \wedge\ z_{k+1} = z) \Rightarrow x\, R^+\, z \right)$$

Take generic x, z and assume that there exist $z_0, \ldots, z_{k+1}$ satisfying the premise of the implication:

$$x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_k\, R\, z_{k+1} \ \wedge\ z_{k+1} = z$$

By the inductive hypothesis, it follows that $x\, R^+\, z_k$. Moreover, from $z_k\, R\, z_{k+1} = z$ we can apply the first inference rule to derive $z_k\, R^+\, z$. Finally, we conclude by applying the second inference rule to $x\, R^+\, z_k$ and $z_k\, R^+\, z$, obtaining $x\, R^+\, z$.

Regarding the second question, the relation R' is just the reflexive and transitive closure of R.

Problems of Chapter 5

5.2

1. It can be readily checked that f is monotone: let us take $S_1, S_2 \in \wp(\mathbb{N})$, with $S_1 \subseteq S_2$; we need to check that $f(S_1) \subseteq f(S_2)$. Let $x \in f(S_1) = S_1 \cap X$. Then $x \in S_1$ and $x \in X$. Since $S_1 \subseteq S_2$, we have also $x \in S_2$ and thus $x \in S_2 \cap X = f(S_2)$.

The case of g is more subtle. Intuitively, the larger is S, the smaller is $\wp(\mathbb{N}) \setminus S$ and consequently $(\wp(\mathbb{N}) \setminus S) \cap X$. Let us take $S_1, S_2 \in \wp(\mathbb{N})$, with $S_1 \subset S_2$, and let $x \in S_2 \setminus S_1$. Then $x \in \wp(\mathbb{N}) \setminus S_1$ and $x \notin \wp(\mathbb{N}) \setminus S_2$. Now if $x \in X$ we have $x \in g(S_1)$ and $x \notin g(S_2)$, contradicting the requirement $g(S_1) \subseteq g(S_2)$. Note that, unless $X = \varnothing$, such a counterexample can always be constructed.

2. Let us take a chain $\{S_i\}_{i \in \mathbb{N}}$ in $\wp(\mathbb{N})$. We need to prove that $f(\bigcup_{i \in \mathbb{N}} S_i) = \bigcup_{i \in \mathbb{N}} f(S_i)$, i.e., that $(\bigcup_{i \in \mathbb{N}} S_i) \cap X = \bigcup_{i \in \mathbb{N}} (S_i \cap X)$. We have

$$\begin{aligned} x \in \left(\bigcup_{i \in \mathbb{N}} S_i \right) \cap X &\Leftrightarrow x \in \left(\bigcup_{i \in \mathbb{N}} S_i \right) \wedge x \in X \\ &\Leftrightarrow \exists k \in \mathbb{N}.\, x \in S_k \wedge x \in X \\ &\Leftrightarrow \exists k \in \mathbb{N}.\, x \in S_k \cap X \\ &\Leftrightarrow x \in \bigcup_{i \in \mathbb{N}} (S_i \cap X) \end{aligned}$$

Since g is in general not monotone, it is not continuous (unless $X = \varnothing$, in which case g is the constant function returning $\varnothing$ and thus trivially monotone and continuous).

3. f is monotone and continuous for any X, while g is monotone and continuous only when $X = \varnothing$.

5.3

1. Let D_1 be the discrete order with two elements 0 and 1. All chains in D_1 are constant (and finite) and all functions $f : D_1 \to D_1$ are monotone and continuous. The identity function $f_1(x) = x$ has two fixpoints but no least fixpoint (as discussed also in Example 5.18).
2. Let $D_2 = D_1$. If we let $f_2(0) = 1$ and $f_2(1) = 0$, then f_2 has no fixpoint.
3. If D_3 is finite, then any chain is finite and any monotone function is continuous. So we must choose D_3 with infinitely many elements. We take D_3 and f_3 as in Example 5.17.

5.4 Let us take $D = \mathbb{N}$ with the usual "less than or equal to" order. As discussed in Chapter 5, it is a partial order with bottom but it is not complete, because, e.g., the chain of even numbers has no upper bound.

1. From what said above, the chain

$$0 \succ 2 \succ 4 \succ 6 \succ \cdots$$

is an infinite descending chain, and thus $\mathscr{D}'$ is not well-founded.

2. The answer is no: if $\mathscr{D}$ is not complete, then $\mathscr{D}'$ is not well-founded. To show this, let us take a chain

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$$

that has no least upper bound (it must exists, because $\mathscr{D}$ is not complete). The chain $\{d_i\}_{i \in \mathbb{N}}$ cannot be finite, as otherwise the maximum element would be the least upper bound. However, it is not necessarily the case that

$$d_0 \succ d_1 \succ d_2 \succ \cdots$$

is an infinite descending chain of $\mathscr{D}'$, because $\{d_i\}_{i\in\mathbb{N}}$ can contain repeated elements. To discard clones, we define the function $next : \mathbb{N} \to \mathbb{N}$ to select the smallest index with which the ith different element appears in the chain (letting d_0 be the 0th element)

$$\begin{aligned} next(0) &\stackrel{\text{def}}{=} 0 \\ next(i+1) &\stackrel{\text{def}}{=} \min\{j \mid d_j \neq d_{j-1} \wedge next(i) < j\} \end{aligned}$$

and take the infinite descending chain

$$d_{next(0)} \succ d_{next(1)} \succ d_{next(2)} \succ \cdots$$

5.5

1. We need to check that $\sqsubseteq$ is reflexive, antisymmetric and transitive.

reflexive:	for any string $\alpha \in V^* \cup V^\infty$ we have $\alpha = \alpha\varepsilon$ and hence $\alpha \sqsubseteq \alpha$;
antisymmetric:	we assume $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \alpha$ and we need to prove that $\alpha = \beta$; let γ and δ such that $\beta = \alpha\gamma$ and $\alpha = \beta\delta$, then $\alpha = \alpha\gamma\delta$: if $\alpha \in V^*$, then it must be $\gamma = \delta = \varepsilon$ and $\alpha = \beta$; if $\alpha \in V^\infty$, from $\beta = \alpha\gamma$ it follows $\beta = \alpha$;
transitive:	we assume $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \gamma$ and we need to prove that $\alpha \sqsubseteq \gamma$; let δ and ω such that $\beta = \alpha\delta$ and $\gamma = \beta\omega$, then $\gamma = \alpha\delta\omega$ and thus $\alpha \sqsubseteq \gamma$.

2. To prove that the order is complete we must show that any chain has a limit. Take

   $$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_n \sqsubseteq \cdots$$

   If the chain is finite, then the greatest element of the chain is the least upper bound. Otherwise, it must be $\alpha_i \in V^*$ for any $i \in \mathbb{N}$ and for any length n we can find a string α_{k_n} in the sequence such that $|\alpha_{k_n}| \geq n$ (if not, the chain would be finite). Then we can construct a string $\alpha \in V^\infty$ such that for any position n in α the nth symbol of α appears in the same position in one of the strings in the chain. In fact we let $\alpha(n) \stackrel{\text{def}}{=} \alpha_{k_n}(n)$ and α is the limit of the chain.
3. The bottom element is the empty string ε, in fact for any $\alpha \in V^* \cup V^\infty$ we have $\varepsilon\alpha = \alpha$ and thus $\varepsilon \sqsubseteq \alpha$.
4. The maximal elements are all and only the strings in V^∞. In fact, on the one hand, taken $\alpha \in V^\infty$ we have

   $$\alpha \sqsubseteq \beta \quad \Leftrightarrow \quad \exists\gamma.\, \beta = \alpha\gamma \quad \Leftrightarrow \quad \beta = \alpha$$

   On the other hand, if $\alpha \in V^*$, then $\alpha \sqsubseteq \alpha a$ and $\alpha \neq \alpha a$.

Problems of Chapter 6

6.1

1. The expression $\lambda x.\ \lambda x.\ x$ is α-convertible to the expressions a, c, e.
2. The expression $((\lambda x.\ \lambda y.\ x)\ y)$ is equivalent to the expressions d and e.

6.3 Let $c'' \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2$. Using the operational semantics we have:

$$\begin{array}{rll}
\langle c, \sigma\rangle \to \sigma' & \nwarrow & \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c'', \sigma''\rangle \to \sigma' \\
& \nwarrow *_{\sigma''=\sigma[^0/_x]} & \langle x = 0, \sigma[^0/_x]\rangle \to \textbf{true}, \quad \langle c_1, \sigma[^0/_x]\rangle \to \sigma' \\
& \nwarrow^* & \langle c_1, \sigma[^0/_x]\rangle \to \sigma' \\
\langle c', \sigma\rangle \to \sigma' & \nwarrow & \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma' \\
& \nwarrow *_{\sigma''=\sigma[^0/_x]} & \langle c_1, \sigma[^0/_x]\rangle \to \sigma'
\end{array}$$

Since both goals reduce to the same goal $\langle c_1, \sigma[^0/_x]\rangle \to \sigma'$, the two commands c and c' are equivalent.

Using the denotational semantics, we have:

$$\begin{aligned}
\mathscr{C}[\![c]\!]\sigma &= \mathscr{C}[\![c'']\!]^*(\mathscr{C}[\![x := 0]\!]\sigma) \\
&= \mathscr{C}[\![c'']\!]^*(\sigma[^0/_x]) \\
&= \mathscr{C}[\![c'']\!](\sigma[^0/_x]) \\
&= (\lambda\sigma'.\ (\mathscr{B}[\![x = 0]\!]\sigma' \to \mathscr{C}[\![c_1]\!]\sigma', \mathscr{C}[\![c_2]\!]\sigma'))(\sigma[^0/_x]) \\
&= \mathscr{B}[\![x = 0]\!]\sigma[^0/_x] \to \mathscr{C}[\![c_1]\!]\sigma[0/x], \mathscr{C}[\![c_2]\!]\sigma[0/x] \\
&= \textbf{true} \to \mathscr{C}[\![c_1]\!]\sigma[0/x], \mathscr{C}[\![c_2]\!]\sigma[0/x] \\
&= \mathscr{C}[\![c_1]\!]\sigma[0/x] \\
\mathscr{C}[\![c']\!]\sigma &= \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![x := 0]\!]\sigma) \\
&= \mathscr{C}[\![c_1]\!]^*(\sigma[0/x]) \\
&= \mathscr{C}[\![c_1]\!](\sigma[0/x]).
\end{aligned}$$

6.4 Let $c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c \textbf{ else skip}$. We have that

$$\begin{aligned}
\Gamma_{b,c}\ \varphi\ \sigma &= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\
\Gamma_{b,c'}\ \varphi\ \sigma &= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c']\!]\sigma), \sigma \\
&= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \mathscr{B}[\![\textbf{skip}]\!]\sigma), \sigma \\
&= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \sigma), \sigma
\end{aligned}$$

Let us show that $\Gamma_{b,c} = \Gamma_{b,c'}$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, then $\Gamma_{b,c}\ \varphi\ \sigma = \sigma = \Gamma_{b,c}\ \varphi\ \sigma$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$, then

$$\begin{aligned}\Gamma_{b,c}\ \varphi\ \sigma &= \varphi^*(\mathscr{C}[\![c]\!]\sigma)\\ \Gamma_{b,c'}\ \varphi\ \sigma &= \varphi^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \sigma)\\ &= \varphi^*(\mathscr{C}[\![c]\!]\sigma)\end{aligned}$$

6.5 We have already seen in Example 6.6 that $\mathscr{C}[\![\textbf{while true do skip}]\!] = \lambda\sigma.\ \bot_{\Sigma_\bot}$.
For the second command we have $\mathscr{C}[\![\textbf{while true do } x := x+1]\!] = \text{fix } \Gamma$, where

$$\begin{aligned}\Gamma\ \varphi\ \sigma &= \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*(\mathscr{C}[\![x := x+1]\!]\sigma), \sigma\\ &= \textbf{true} \to \varphi^*(\mathscr{C}[\![x := x+1]\!]\sigma), \sigma\\ &= \varphi^*(\mathscr{C}[\![x := x+1]\!]\sigma)\\ &= \varphi^*(\sigma[\sigma(x)+1/x])\\ &= \varphi(\sigma[\sigma(x)+1/x])\end{aligned}$$

Let us compute the first elements of the chain $\{\varphi_n\}_{n\in\mathbb{N}}$ with $\varphi_n = \Gamma^n\ \bot_{\Sigma\to\Sigma_\bot}$:

$$\begin{aligned}\varphi_0\ \sigma &= \bot_{\Sigma_\bot}\\ \varphi_1\ \sigma &= \Gamma\ \varphi_0\ \sigma\\ &= \varphi_0(\sigma[\sigma(x)+1/x])\\ &= (\lambda\sigma.\ \bot_{\Sigma_\bot})(\sigma[\sigma(x)+1/x])\\ &= \bot_{\Sigma_\bot}\end{aligned}$$

Since $\varphi_1 = \varphi_0$ we have reached the fixpoint and have $\mathscr{C}[\![\textbf{while true do } x := x+1]\!] = \lambda\sigma.\ \bot_{\Sigma_\bot}$.

6.6 We have immediately $\mathscr{C}[\![x := 0]\!]\sigma = \sigma[0/x]$.
Moreover, we have $\mathscr{C}[\![\textbf{while } x \neq 0 \textbf{ do } x := 0]\!] = \text{fix } \Gamma$, where

$$\begin{aligned}\Gamma\ \varphi\ \sigma &= \mathscr{B}[\![x \neq 0]\!]\sigma \to \varphi^*(\mathscr{C}[\![x := 0]\!]\sigma), \sigma\\ &= \sigma(x) \neq 0 \to \varphi^*(\sigma[0/x]), \sigma\\ &= \sigma(x) \neq 0 \to \varphi(\sigma[0/x]), \sigma\end{aligned}$$

Let us compute the first elements of the chain $\{\varphi_n\}_{n\in\mathbb{N}}$ with $\varphi_n = \Gamma^n\ \bot_{\Sigma\to\Sigma_\bot}$:

$$
\begin{aligned}
\varphi_0\ \sigma &= \bot_{\Sigma_\bot} \\
\varphi_1\ \sigma &= \Gamma\ \varphi_0\ \sigma \\
&= \sigma(x) \neq 0 \to \varphi_0(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to \bot_{\Sigma_\bot}\,,\sigma \\
\varphi_2\ \sigma &= \Gamma\ \varphi_1\ \sigma \\
&= \sigma(x) \neq 0 \to \varphi_1(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\lambda\sigma'.\ \sigma'(x) \neq 0 \to \bot_{\Sigma_\bot}\,,\sigma')(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\sigma[0/x](x) \neq 0 \to \bot_{\Sigma_\bot}\,,\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\mathbf{false} \to \bot_{\Sigma_\bot}\,,\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x]\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x]\,,\sigma[0/x] \\
&= \sigma[0/x] \\
\varphi_3\ \sigma &= \Gamma\ \varphi_2\ \sigma \\
&= \sigma(x) \neq 0 \to \varphi_2(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\lambda\sigma'.\ \sigma'[0/x])(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x][0/x]\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x]\,,\sigma[0/x] \\
&= \sigma[0/x]
\end{aligned}
$$

Note in fact that, when $\sigma(x) \neq 0$ is false, then $\sigma = \sigma[0/x]$.

Since $\varphi_3 = \varphi_2$ we have reached the fixpoint and have $\mathscr{C}[\![\mathbf{while}\ x \neq 0\ \mathbf{do}\ x := 0]\!] = \lambda\sigma.\ \sigma[0/x]$.

We conclude by observing that since φ_2 is a maximal element of its domain, it must be already the lub of the chain, namely the fixpoint. Thus it is not necessary to compute φ_3.

6.10

1.

$$\frac{\langle c,\sigma\rangle \to \sigma' \quad \langle b,\sigma'\rangle \to \mathbf{false}}{\langle \mathbf{do}\ c\ \mathbf{undoif}\ b,\sigma\rangle \to \sigma'}\ (\text{do}) \qquad \frac{\langle c,\sigma\rangle \to \sigma' \quad \langle b,\sigma'\rangle \to \mathbf{true}}{\langle \mathbf{do}\ c\ \mathbf{undoif}\ b,\sigma\rangle \to \sigma}\ (\text{undo})$$

2.

$$\mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\,\sigma \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*\,(\mathscr{C}[\![c]\!]\,\sigma) \to^*\ \sigma\,,\mathscr{C}[\![c]\!]\,\sigma$$

where $\mathscr{B}[\![b]\!]^* : \Sigma_\bot \to \mathbb{B}_\bot$ denotes the lifted version of the interpretation functions for boolean expressions (as c can diverge) and $t \to^* t_0,t_1$ denotes the lifted version of the conditional operator, such that it returns $\bot_{\Sigma_\bot}$ when t is $\bot_{\mathbb{B}_\bot}$.

3. First we extend the proof of completeness by rule induction. We recall that:

$$P\left(\langle c,\sigma\rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\,\sigma = \sigma'$$

do: we assume that $\langle b, \sigma' \rangle \to \textbf{false}$ and $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$. We need to prove that

$$P\big(\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'\big) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma'$$

From $\langle b, \sigma' \rangle \to \textbf{false}$ it follows $\mathscr{B}[\![b]\!](\sigma') = \textbf{false}$. We have

$$\begin{aligned}
\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma &\stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to^* \sigma, \mathscr{C}[\![c]\!]\sigma \\
&= \mathscr{B}[\![b]\!]^*\sigma' \to^* \sigma, \sigma' \\
&= \mathscr{B}[\![b]\!]\sigma' \to^* \sigma, \sigma' \\
&= \textbf{false} \to^* \sigma, \sigma' \\
&= \textbf{false} \to \sigma, \sigma' \\
&= \sigma'.
\end{aligned}$$

undo: we assume that $\langle b, \sigma' \rangle \to \textbf{true}$ and $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$. We need to prove that

$$P(\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma$$

From $\langle b, \sigma' \rangle \to \textbf{true}$ it follows $\mathscr{B}[\![b]\!](\sigma') = \textbf{true}$. We have

$$\begin{aligned}
\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma &\stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to^* \sigma, \mathscr{C}[\![c]\!]\sigma \\
&= \mathscr{B}[\![b]\!]^*\sigma' \to^* \sigma, \sigma' \\
&= \mathscr{B}[\![b]\!]\sigma' \to^* \sigma, \sigma' \\
&= \textbf{true} \to^* \sigma, \sigma' \\
&= \textbf{true} \to \sigma, \sigma' \\
&= \sigma.
\end{aligned}$$

Finally, we extend the proof of correctness by structural induction. We assume

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![c]\!]\sigma = \sigma' \Rightarrow \langle c, \sigma \rangle \to \sigma'$$

and we want to prove that

$$P(\textbf{do } c \textbf{ undoif } b) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma' \Rightarrow \langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'$$

Let us take σ and σ' such that $\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma'$. We need to prove that $\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'$. Since $\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma'$ it must be $\mathscr{C}[\![c]\!]\sigma = \sigma''$ for some $\sigma'' \neq \bot_{\Sigma_\bot}$ and by inductive hypothesis $\langle c, \sigma \rangle \to \sigma''$. We distinguish two cases.

$\mathscr{B}[\![b]\!]\sigma'' = \textbf{false}$: then $\sigma' = \sigma''$ and $\langle b, \sigma'' \rangle \to \textbf{false}$. Since $\langle c, \sigma \rangle \to \sigma''$ we apply rule (do) to derive $\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'' = \sigma'$.

$\mathscr{B}[\![b]\!]\,\sigma'' = \mathbf{true}$: then $\sigma' = \sigma$ and $\langle b, \sigma''\rangle \to \mathbf{true}$. Since $\langle c, \sigma\rangle \to \sigma''$ we apply rule (undo) to conclude that $\langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma\rangle \to \sigma$.

Problems of Chapter 7

7.2

$$\mathbf{rec}\ \underset{\tau_2\to int}{f}\ .\ \lambda\ \underset{\tau*int}{x}\ .\ \mathbf{if}\ \underbrace{\mathbf{snd}(\ \underset{\tau*int}{x}\ )}_{int}\ \mathbf{then}\ \underset{int}{1}\ \mathbf{else}\ \underset{\tau_2\to int}{f}\ (\ \underbrace{\mathbf{fst}(x)}_{int\to\tau_1}\ ,\ (\ \underbrace{\mathbf{fst}(x)}_{\tau=int\to\tau_1}\ \underbrace{\mathbf{snd}(x)}_{int}\ )\ )$$

τ_1; $\tau_2 = (int\to\tau_1)*\tau_1$; int; $(\tau*int)\to int$

From which we must have $\tau_2 \to int = (\tau * int) \to int$, i.e., $\tau_2 = (\tau * int)$. But since $\tau_2 = (int \to \tau_1) * \tau_1$, it must be $\tau = (int \to \tau_1)$ and $int = \tau_1$. Summing up, we have $\tau_1 = int$, $\tau = int \to int$ and $\tau_2 = (int \to int) * int$ and the principal type of t is $((int \to int) * int) \to int$.

7.3

1. We let $\tau = int * (int * (int * int))$ be the type of a list of integers with three elements (the last element of type *int* is 0 and it marks the end of the list) and we define

$$t \stackrel{\text{def}}{=} \lambda\ \underset{\tau}{\ell}.\ \mathbf{fst}(\mathbf{snd}(\ \underbrace{\mathbf{snd}(\underset{\tau}{\ell})}_{\tau}\ ))$$

$int*(int*int)$; $int*int$; int; $\tau\to int$

Let $L = (n_1, (n_2, (n_3, 0))) : \tau$ be a generic list of integers with three elements. Now we check that $(t\ L) \to n_3$:

$$\begin{array}{rl}
(t\ L) \to c & \nwarrow\ t \to \lambda x.\, t',\quad t'[{}^{L}/_{x}] \to c \\
\nwarrow^{*}_{x=\ell,\ t'=\mathbf{fst}(\mathbf{snd}(\mathbf{snd}(\ell)))} & \mathbf{fst}(\mathbf{snd}(\mathbf{snd}(L))) \to c \\
& \nwarrow\ \mathbf{snd}(\mathbf{snd}(L)) \to (t_1,t_2),\quad t_1 \to c \\
& \nwarrow\ \mathbf{snd}(L) \to (t_3,t_4),\quad t_4 \to (t_1,t_2),\quad t_1 \to c \\
& \nwarrow\ L \to (t_5,t_6),\quad t_6 \to (t_3,t_4),\quad t_4 \to (t_1,t_2),\quad t_1 \to c \\
\nwarrow_{t_5=n_1,\ t_6=(n_2,(n_3,0))} & (n_2,(n_3,0)) \to (t_3,t_4),\quad t_4 \to (t_1,t_2),\quad t_1 \to c \\
\nwarrow_{t_3=n_2,\ t_4=(n_3,0)} & (n_3,0) \to (t_1,t_2),\quad t_1 \to c \\
\nwarrow_{t_1=n_3,\ t_2=0} & n_3 \to c \\
\nwarrow_{c=n_3} & \square
\end{array}$$

2. The answer is negative. In fact a generic list of k integers has a type that depends on the length of the list itself and we do not have polymorphic functions in HOFL. The natural candidate

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{snd}(x)\ \textbf{then}\ \textbf{fst}(x)\ \textbf{else}\ f\ (\textbf{snd}(x))$$

is not typable, in fact we have

$$\underset{int\to\tau}{f}\ .\ \lambda\ \underset{\tau*int}{x}\ .\ \textbf{if}\ \textbf{snd}(\underbrace{\underset{\tau*int}{x}}_{int})\ \textbf{then}\ \underbrace{\textbf{fst}(x)}_{\tau}\ \textbf{else}\ \underbrace{\underset{int\to\tau}{f}\ \underbrace{(\textbf{snd}(x))}_{int}}_{\tau}$$

From which we must have $int \to \tau = (\tau * int) \to \tau$, i.e., $int = (\tau * int)$, which is not possible.

7.4

1. We have:

$$t_1 \stackrel{\text{def}}{=} \lambda\ \underset{int}{x}\ .\ \lambda\ \underset{\tau_1}{y}\ .\ \underbrace{\underbrace{\underset{int}{x}\ \underset{int}{3}}_{int}}_{\tau_1\to int}$$

$$t_2 \stackrel{\text{def}}{=} \lambda\ \underset{int*\tau_2}{z}\ .\ \underbrace{\textbf{fst}(\underbrace{\underset{int*\tau_2}{z}}_{int*\tau_2})}_{int}\ \underbrace{+\ \underset{int}{3}}_{int}$$

2. Assume $\tau_1 = \tau_2 = \tau$ with $c : \tau$ in canonical form. We compute the canonical forms of $((t_1\ 1)\ c)$ and $(t_2\ (1,c))$ as follows:

$$\begin{aligned}
((t_1\ 1)\ c) \to c_1 &\nwarrow (t_1\ 1) \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1\\
&\nwarrow t_1 \to \lambda x'.\ t'',\quad t''[^{1}/_{x'}] \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1\\
&\nwarrow_{x'=x,\ t''=\lambda y.\ x+3} \lambda y.\ 1+3 \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1\\
&\nwarrow_{y'=y,\ t'=1+3} 1+3 \to c_1\\
&\nwarrow_{c_1=n_1\underline{+}n_2} 1 \to n_1,\quad 3 \to n_2\\
&\nwarrow^{*}_{n_1=1,\ n_2=3} \square
\end{aligned}$$

Thus $c_1 = n_1 \underline{+} n_2 = 1 \underline{+} 3 = 4$ is the canonical form of $((t_1\ 1)\ c)$.

$$\begin{aligned}
(t_2\ (1,c)) \to c_2 &\nwarrow t_2 \to \lambda z'.\ t',\quad t'[^{(1,c)}/_{z'}] \to c_2\\
&\nwarrow_{z'=z,\ t'=\textbf{fst}(z)+3} \textbf{fst}((1,c)) + 3 \to c_2\\
&\nwarrow_{c_2=n_1\underline{+}n_2} \textbf{fst}((1,c)) \to n_1,\quad 3 \to n_2\\
&\nwarrow (1,c) \to (t'',t'''),\quad t'' \to n_1,\quad 3 \to n_2\\
&\nwarrow_{t''=1,\ t'''=c} 1 \to n_1,\quad 3 \to n_2\\
&\nwarrow^{*}_{n_1=1,\ n_2=3} \square
\end{aligned}$$

Thus $c_2 = n_1 \underline{+} n_2 = 1 \underline{+} 3 = 4$ is the canonical form also of $(t_2\ (1,c))$.

7.5 We find the principal type of *map*:

$$map \stackrel{\text{def}}{=} \lambda \underset{\tau_1\to\tau}{f}.\ \lambda \underset{\tau_1*\tau_1}{x}.\ ((\underset{\tau_1\to\tau}{f}\ \underbrace{\mathbf{fst}(\underset{\tau_1*\tau_2}{x})}_{\tau_1}), (\underset{\tau_1\to\tau}{f}\ \underbrace{\mathbf{snd}(\underset{\tau_1*\tau_2}{x})}_{\tau_2=\tau_1}))$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{\tau}\qquad\underbrace{\qquad\qquad\qquad\qquad}_{\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\tau*\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{(\tau_1*\tau_1)\to(\tau,\tau)}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{(\tau_1\to\tau)\to(\tau_1*\tau_1)\to(\tau,\tau)}$$

We now compute the canonical form of the term $((map\ t)\ (1,2))$ where $t \stackrel{\text{def}}{=} \lambda x.\ 2\times x$:

$$\begin{aligned}
((map\ t)\ (1,2)) \to c \quad & \nwarrow (map\ t) \to \lambda y.t',\quad t'[{}^{(1,2)}/_{y}] \to c \\
& \nwarrow map \to \lambda g.t'',\quad t''[{}^{t}/_{g}] \to \lambda y.t',\quad t'[{}^{(1,2)}/_{y}] \to c \\
& \nwarrow_{g=f,\ t''=\ldots} \lambda x.\ ((t\ \mathbf{fst}(x)), (t\ \mathbf{snd}(x))) \to \lambda y.t',\quad t'[{}^{(1,2)}/_{y}] \to c \\
& \nwarrow_{y=x,\ t'=\ldots} ((t\ \mathbf{fst}((1,2))), (t\ \mathbf{snd}((1,2)))) \to c \\
\nwarrow_{c=((t\ \mathbf{fst}((1,2))),(t\ \mathbf{snd}((1,2))))} & \ \square
\end{aligned}$$

So the canonical form is $c = (((\lambda x.\ 2\times x)\ \mathbf{fst}((1,2))), ((\lambda x.\ 2\times x)\ \mathbf{snd}((1,2))))$.

Problems of Chapter 8

8.4 We prove the monotonicity of the lifting operator $(\cdot)^* : [D\to E] \to [D_\perp \to E]$. Let us take two continuous functions $f, g \in [D\to E]$ such that $f \sqsubseteq_{D\to E} g$. We want to prove that $f^* \sqsubseteq_{D_\perp\to E} g^*$. So we need to prove that for any $x \in D_\perp$ we have $f^*(x) \sqsubseteq_E g^*(x)$. We have two possibilities:

- if $x = \perp_{D_\perp}$, then $f^*(\perp_{D_\perp}) = \perp_E = g^*(\perp_{D_\perp})$;
- if $x = \lfloor d \rfloor$ for some $d \in D$, we have $f^*(\lfloor d \rfloor) = f(d) \sqsubseteq g(d) = g^*(\lfloor d \rfloor)$, because $f \sqsubseteq_{D\to E} g$ by hypothesis.

8.5 We prove that the function apply $: [D\to E]\times D \to E$ is monotone. Let us take two continuous functions $f_1, f_2 \in [D\to E]$ and two elements $d_1, d_2 \in D$ such that $(f_1, d_1) \sqsubseteq_{[D\to E]\times D} (f_2, d_2)$, we want to prove that apply $(f_1, d_1) \sqsubseteq_E$ apply (f_2, d_2). By definition of the cartesian product domain, $(f_1, d_1) \sqsubseteq_{[D\to E]\times D} (f_2, d_2)$ means that $f_1 \sqsubseteq_{[D\to E]} f_2$ and $d_1 \sqsubseteq_D d_2$. Then, we have:

$$\begin{array}{rcll}
\text{apply}\ (f_1, d_1) & = & f_1(d_1) & \text{(by definition of apply)} \\
& \sqsubseteq_E & f_1(d_2) & \text{(by monotonicity of } f_1\text{)} \\
& \sqsubseteq_E & f_2(d_2) & \text{(because } f_1 \sqsubseteq_{[D\to E]} f_2\text{)} \\
& = & \text{apply}\ (f_2, d_2) & \text{(by definition of apply).}
\end{array}$$

8.6 Let $\mathbf{F}_f = \{d \mid d = f(d)\} \subseteq D$ be the set of fixpoints of $f : D \to D$. It is immediate that $\mathbf{F}_f$ is a PO, because it is a subset of the partial order D from which it inherits

the order relation. It remains to be proved that it is complete. Take a chain $\{d_i\}_{i\in\mathbb{N}}$ in $\mathbf{F}_f$. Since $\mathbf{F}_f \subseteq D$ and D is a CPO, the chain $\{d_i\}_{i\in\mathbb{N}}$ has a limit $d = \bigsqcup_{i\in\mathbb{N}} d_i$ in D. We want to prove that $d \in \mathbf{F}_f$, i.e., that $d = f(d)$. We note that for any $i \in \mathbb{N}$ we have $d_i = f(d_i)$, because $d_i \in \mathbf{F}_f$. Since f is continuous, we have:

$$f(d) = f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} f(d_i) = \bigsqcup_{i\in\mathbb{N}} d_i = d.$$

8.8 We divide the proof in two parts: first we show that $f \sqsubseteq g$ implies $f \preceq g$ and then that $f \preceq g$ implies $f \sqsubseteq g$.

For the first implication, suppose that $f \sqsubseteq g$. Taken any two elements $d_1, d_2 \in D$ such that $d_1 \sqsubseteq_D d_2$ we want to prove that $f(d_1) \sqsubseteq_E g(d_2)$. From the monotonicity of f we have $f(d_1) \sqsubseteq_E f(d_2)$ and by the hypothesis $f \sqsubseteq g$ it follows that $f(d_2) \sqsubseteq_E g(d_2)$; thus, $f(d_1) \sqsubseteq_E f(d_2) \sqsubseteq_E g(d_2)$.

For the second implication, suppose $f \preceq g$. We want to prove that for any element $d \in D$ we have $f(d) \sqsubseteq_E g(d)$. But this is immediate, because by reflexivity we have $d \sqsubseteq_D d$ and thus $f(d) \sqsubseteq_E g(d)$ by definition of $\preceq$.

Problems of Chapter 9

9.1 We show that t is typable:

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ \underset{int\to int}{f}\ .\ \lambda\ \underset{int}{x}\ .\ \mathbf{if}\ \underset{int}{x}\ \mathbf{then}\ \underset{int}{0}\ \mathbf{else}\ (\ \underset{int\to int}{f}\ (\underset{int}{x}) \times\ \underset{int\to int}{f}\ (\underset{int}{x}))$$

$$int \to int$$

So we conclude $t : int \to int$.

The canonical form is readily obtained by unfolding once the recursive definition:

$$t \to c \qquad \nwarrow \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ (t(x) \times t(x)) \to c$$
$$\nwarrow_{c=\lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ (t(x)\times t(x))} \square$$

Finally, the denotational semantics is computed as follows:

$$
\begin{aligned}
[\![t]\!]\rho &= \text{fix } \lambda d_f.\ [\![\lambda x.\ \textbf{if } x \textbf{ then } 0 \textbf{ else } (f(x)\times f(x))]\!]\rho[^{d_f}/_f] \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ [\![\textbf{if } x \textbf{ then } 0 \textbf{ else } (f(x)\times f(x))]\!] \underbrace{\rho[^{d_f}/_f, ^{d_x}/_x]}_{\rho'} \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ Cond([\![x]\!]\rho', [\![0]\!]\rho', [\![f(x)\times f(x)]\!]\rho') \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, [\![f(x)]\!]\rho' \underline{\times}_\perp [\![f(x)]\!]\rho') \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\textbf{let } \varphi \Leftarrow d_f.\ \varphi(d_x)) \underline{\times}_\perp (\textbf{let } \varphi \Leftarrow d_f.\ \varphi(d_x))) \rfloor
\end{aligned}
$$

because

$$
\begin{aligned}
[\![f(x)]\!]\rho' &= \textbf{let } \varphi \Leftarrow [\![f]\!]\rho'.\ \varphi([\![x]\!]\rho') \\
&= \textbf{let } \varphi \Leftarrow d_f.\ \varphi(d_x)
\end{aligned}
$$

Let us compute the fixpoint by successive approximations:

$$
\begin{aligned}
f_0 &= \perp_{(V_{int}\to int)_\perp} \\
f_1 &= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\textbf{let } \varphi \Leftarrow f_0.\ \varphi(d_x)) \underline{\times}_\perp (\textbf{let } \varphi \Leftarrow f_0.\ \varphi(d_x))) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\perp_{(V_{int})_\perp}) \underline{\times}_\perp (\perp_{(V_{int})_\perp})) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}) \rfloor \\
f_2 &= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\textbf{let } \varphi \Leftarrow f_1.\ \varphi(d_x)) \underline{\times}_\perp (\textbf{let } \varphi \Leftarrow f_1.\ \varphi(d_x))) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp})) \underline{\times}_\perp (Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}))) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\perp_{(V_{int})_\perp}) \underline{\times}_\perp (\perp_{(V_{int})_\perp})) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}) \rfloor \\
&= f_1
\end{aligned}
$$

So we have reached the fixpoint and

$$[\![t]\!]\rho = \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}) \rfloor$$

9.9

1. Assume $t_1 : \tau$. We have

$$t_2 \stackrel{\text{def}}{=} \underbrace{\lambda \underbrace{x}_{\tau_1}.\ \underbrace{(\underbrace{t_1}_{\tau_1\to\tau_2}\ \underbrace{x}_{\tau_1})}_{\tau_2}}_{\tau_1\to\tau_2}$$

   Unless $\tau = \tau_1 \to \tau_2$ the pre-term t_2 is not typable.
2. Let us compute the denotational semantics of t_2:

$$\begin{aligned}
[\![t_2]\!]\rho &= \left\lfloor \lambda d_x.\ [\![t_1\ x]\!]\rho[{}^{d_x}/_x] \right\rfloor \\
&= \left\lfloor \lambda d_x.\ \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho[{}^{d_x}/_x].\ \varphi([\![x]\!]\rho[{}^{d_x}/_x]) \right\rfloor \\
&= \left\lfloor \lambda d_x.\ \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho[{}^{d_x}/_x].\ \varphi(d_x) \right\rfloor
\end{aligned}$$

Suppose $x \notin \mathrm{fv}(t_1)$. Then we have $\forall y \in \mathrm{fv}(t_1).\ \rho(y) = \rho[{}^{d_x}/_x](y)$ and thus by Theorem 9.5 we have $[\![t_1]\!]\rho[{}^{d_x}/_x] = [\![t_1]\!]\rho$.
Now, if $[\![t_1]\!]\rho = \bot_{(V_\tau)_\bot}$, then $[\![t_2]\!]\rho = \left\lfloor \lambda d_x.\ \bot_{(V_{\tau_2})_\bot} \right\rfloor \neq [\![t_1]\!]\rho$.
Otherwise, it must be $[\![t_1]\!]\rho = \lfloor f \rfloor$ for some $f \in V_{\tau_1 \to \tau_2}$ and hence $[\![t_2]\!]\rho = \lfloor \lambda d_x.\ f\ d_x \rfloor = \lfloor f \rfloor = [\![t_1]\!]\rho$.

9.10

1. Let us compute the principal types for t_1 and t_2:

$t_1 \stackrel{\text{def}}{=} \lambda\, \underset{\tau_1}{x}.\ \textbf{rec}\ \underset{int}{y}.\ \underset{int}{(}\underset{int}{y} + \underset{int}{1}\underset{int}{)}$ — $\tau_1 \to int$

$t_2 \stackrel{\text{def}}{=} \textbf{rec}\ \underset{\tau_2 \to int}{y}.\ \lambda\, \underset{\tau_2}{x}.\ \underset{int}{(}\underset{\tau_2 \to int}{y}\ \underset{\tau_2}{x}\underset{int}{)} + \underset{int}{2}$ — $\tau_2 \to int$

Therefore t_1 and t_2 have the same type if and only if $\tau_1 = \tau_2$.

2. Let us compute the denotational semantics of t_1:

$$\begin{aligned}
[\![t_1]\!]\rho &= \lfloor \lambda d_x.\ [\![\textbf{rec}\ y.\ y+1]\!]\rho[{}^{d_x}/_x] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ [\![y+1]\!]\rho[{}^{d_x}/_x, {}^{d_y}/_y] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ [\![y]\!]\rho[{}^{d_x}/_x, {}^{d_y}/_y] \pm_\bot [\![1]\!]\rho[{}^{d_x}/_x, {}^{d_y}/_y] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ d_y \pm_\bot \lfloor 1 \rfloor \rfloor
\end{aligned}$$

We need to compute the fixpoint $\mathrm{fix}\ \lambda d_y.\ d_y \pm_\bot \lfloor 1 \rfloor$:

$$\begin{aligned}
d_0 &= \bot_{(V_{int})_\bot} \\
d_1 &= d_0 \pm_\bot \lfloor 1 \rfloor = \bot_{(V_{int})_\bot} = d_0
\end{aligned}$$

From which it follows

$$[\![t_1]\!]\rho = \lfloor \lambda d_x.\ \bot_{(V_{int})_\bot} \rfloor = \lfloor \bot_{(V_{\tau \to int})} \rfloor$$

Let us now turn the attention to t_2:

$$\begin{aligned}
[\![t_2]\!]\rho &= \text{fix } \lambda d_y.\ [\![\lambda x.\ (y\ x)+2]\!]\rho[{}^{d_y}/_{y}] \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ [\![(y\ x)+2]\!]\underbrace{\rho[{}^{d_y}/_{y}, {}^{d_x}/_{x}]}_{\rho'}\rfloor \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ [\![y\ x]\!]\rho' \pm_\perp [\![2]\!]\rho'\rfloor \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow [\![y]\!]\rho'.\ \varphi([\![x]\!]\rho')) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow d_y.\ \varphi(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor
\end{aligned}$$

Let us compute the fixpoint:

$$\begin{aligned}
f_0 &= \perp_{(V_{\tau\to int})_\perp} \\
f_1 &= \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow f_0.\ \varphi(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ (\perp_{(V_{int})_\perp}) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ \perp_{(V_{int})_\perp}\rfloor \\
&= \lfloor \perp_{(V_{\tau\to int})}\rfloor \\
f_2 &= \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow f_1.\ \varphi(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ (\perp_{(V_{\tau\to int})}(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ (\perp_{(V_{int})_\perp}) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ \perp_{(V_{int})_\perp}\rfloor \\
&= \lfloor \perp_{(V_{\tau\to int})}\rfloor \\
&= f_1
\end{aligned}$$

So we have computed the fixpoint and got

$$[\![t_2]\!]\rho = \lfloor \perp_{(V_{\tau\to int})}\rfloor = [\![t_1]\!]\rho.$$

9.15 Let us try to change the denotational semantics of the conditional construct of HOFL by defining:

$$[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho \stackrel{\text{def}}{=} \mathit{Cond}'([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

where

$$\mathit{Cond}'(x, d_0, d_1) = \begin{cases} d_0 & \text{if } x = \lfloor n\rfloor \text{ for some } n \in \mathbb{Z} \\ d_1 & \text{if } x = \perp_{(V_{int})_\perp}. \end{cases}$$

The problem is that the newly defined operation Cond' is not monotone (and thus not continuous)! To see this, remind that $\perp_{(V_{int})_\perp} \sqsubseteq \lfloor 1\rfloor$ and take any d_0, d_1: we should have $\mathit{Cond}'(\perp_{(V_{int})_\perp}, d_0, d_1) \sqsubseteq \mathit{Cond}'(\lfloor 1\rfloor, d_0, d_1)$. However, if we take d_0, d_1 such that $d_1 \not\sqsubseteq d_0$ it follows that:

$$\mathit{Cond}'(\perp_{(V_{int})_\perp}, d_0, d_1) = d_1 \not\sqsubseteq d_0 = \mathit{Cond}'(\lfloor 1\rfloor, d_0, d_1)$$

For a concrete example, take $d_1 = \lfloor 1\rfloor$ and $d_0 = \lfloor 0\rfloor$.

At the level of HOFL syntax, the previous cases arise when considering, e.g., the terms $t_1 \stackrel{\text{def}}{=} \textbf{if } (\textbf{rec } x.\ x) \textbf{ then } 0 \textbf{ else } 1$ and $t_2 \stackrel{\text{def}}{=} \textbf{if } 1 \textbf{ then } 0 \textbf{ else } 1$, as

$$[\![t_1]\!]\rho = \lfloor 1 \rfloor \not\sqsubseteq \lfloor 0 \rfloor = [\![t_2]\!]\rho$$

As a consequence, typable terms such as

$$t \stackrel{\text{def}}{=} \lambda x.\ \textbf{if } x \textbf{ then } 0 \textbf{ else } 1 \ :\ \mathit{int} \to \mathit{int}$$

would not be assigned a semantics in $(V_{\mathit{int}\to\mathit{int}})_\perp$ because the function $[\![t]\!]\rho$ would not be continuous.

Problems of Chapter 10

10.1 Let us check the type of t_1 and t_2:

$$\underbrace{\textbf{rec } \underbrace{f}_{\tau_2\to\tau_1}\ .\ \lambda \underbrace{x}_{\tau_2}\ .\ ((\lambda \underbrace{y}_{\tau_1}\ .\ \underbrace{1}_{\mathit{int}})\ (\underbrace{f}_{\tau_2\to\tau_1}\ \underbrace{x}_{\tau_2}))}_{\tau_1\to\mathit{int}\quad\tau_1}}_{\mathit{int}}}_{\tau_2\to\mathit{int}}}_{\tau_2\to\tau_1=\tau_2\to\mathit{int}} \qquad \underbrace{\lambda \underbrace{x}_{\tau}\ .\ \underbrace{1}_{\mathit{int}}}_{\tau\to\mathit{int}}$$

So it must be $\tau_1 = \mathit{int}$ and the terms have the same type if $\tau_2 = \tau$.

The denotational semantics of t_1 requires the computation of the fixpoint:

$$\begin{aligned}
[\![t_1]\!]\rho &= \text{fix } \lambda d_f.\ [\![\lambda x.\ ((\lambda y.\ 1)\ (f\ x))]\!]\rho[^{d_f}/_f] \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ [\![((\lambda y.\ 1)\ (f\ x))]\!]\underbrace{\rho[^{d_f}/_f, {}^{d_x}/_x]}_{\rho'}\rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow [\![\lambda y.\ 1]\!]\rho'.\ \varphi([\![f\ x]\!]\rho'))\rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow \lfloor \lambda d_y.\ \lfloor 1 \rfloor \rfloor.\ (\varphi(\textbf{let } \varphi' \Leftarrow d_f.\ \varphi'(d_x))))\rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ ((\lambda d_y.\ \lfloor 1 \rfloor)(\textbf{let } \varphi' \Leftarrow d_f.\ \varphi'(d_x)))\rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor \\
f_0 &= \perp_{(V_{\tau\to\mathit{int}})_\perp} \\
f_1 &= \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor
\end{aligned}$$

We can stop the calculation of the fixpoint, as we have reached a maximal element. Thus $[\![t_1]\!]\rho = \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor$. For t_2 we have directly:

$$\begin{aligned}\llbracket t_2 \rrbracket \rho &= \lfloor \lambda d_x.\ \llbracket 1 \rrbracket \rho[^{d_x}/_x] \rfloor \\ &= \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor \\ &= \llbracket t_1 \rrbracket \rho\end{aligned}$$

To show that the canonical forms are different, we note that t_2 is already in canonical form, while for t_1 we have:

$$t_1 \to c_1 \qquad \nwarrow \lambda x.\ ((\lambda y.\ 1)\ (t_1\ x)) \to c_1$$
$$\nwarrow_{c_1 = \lambda x.\ ((\lambda y.\ 1)\ (t_1\ x))} \square$$

10.2

1. We compute the denotational semantics of *map* and of $t \stackrel{\text{def}}{=} (map\ \lambda z.\ z)$:

$$\begin{aligned}\llbracket map \rrbracket \rho &= \lfloor \lambda d_f.\ \llbracket \lambda x.\ ((f\ \mathbf{fst}(x)), (f\ \mathbf{snd}(x))) \rrbracket \rho[^{d_f}/_f] \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \llbracket ((f\ \mathbf{fst}(x)), (f\ \mathbf{snd}(x))) \rrbracket \underbrace{\rho[^{d_f}/_f, {}^{d_x}/_x]}_{\rho'} \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor (\llbracket (f\ \mathbf{fst}(x)) \rrbracket \rho', \llbracket (f\ \mathbf{snd}(x)) \rrbracket \rho') \rfloor \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow \llbracket f \rrbracket \rho'.\ \varphi_1(\llbracket \mathbf{fst}(x) \rrbracket \rho')), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow \llbracket f \rrbracket \rho'.\ \varphi_2(\llbracket \mathbf{snd}(x) \rrbracket \rho'))) \rfloor \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow d_f.\ \varphi_1(\mathbf{let}\ d_1 \Leftarrow \llbracket x \rrbracket \rho'.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow d_f.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow \llbracket x \rrbracket \rho'.\ \pi_2\ d_2))) \rfloor \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow d_f.\ \varphi_1(\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow d_f.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2))) \rfloor \rfloor \rfloor \\ \llbracket t \rrbracket \rho &= \mathbf{let}\ \varphi \Leftarrow \llbracket map \rrbracket \rho.\ \varphi(\llbracket \lambda z.\ z \rrbracket \rho) \\ &= \mathbf{let}\ \varphi \Leftarrow \llbracket map \rrbracket \rho.\ \varphi(\lfloor \lambda d_z.\ \llbracket z \rrbracket \rho[^{d_z}/_z] \rfloor) \\ &= \mathbf{let}\ \varphi \Leftarrow \llbracket map \rrbracket \rho.\ \varphi(\lfloor \lambda d_z.\ d_z \rfloor) \\ &= \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow \lfloor \lambda d_z.\ d_z \rfloor.\ \varphi_1(\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow \lfloor \lambda d_z.\ d_z \rfloor.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2))) \rfloor \rfloor \\ &= \lfloor \lambda d_x.\ \lfloor (((\lambda d_z.\ d_z)(\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1)), \\ &\qquad\qquad ((\lambda d_z.\ d_z)(\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2))) \rfloor \rfloor \\ &= \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1), (\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2)) \rfloor \rfloor\end{aligned}$$

2. It suffices to take $t_1 \stackrel{\text{def}}{=} 1+1$ and $t_2 \stackrel{\text{def}}{=} 2$. It can be readily checked that

$$\llbracket (t_1, t_2) \rrbracket \rho = \lfloor (\lfloor 2 \rfloor, \lfloor 2 \rfloor) \rfloor = \llbracket (t_2, t_1) \rrbracket \rho.$$

Letting $t_0 \stackrel{\text{def}}{=} (map\ \lambda z.\ z)$, we have that the terms $(t_0\ (t_1, t_2))$ and $(t_0\ (t_2, t_1))$ have the same denotational semantics

$$\begin{aligned}
[\![t_0\ (t_1,t_2)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi([\![(t_1,t_2)]\!]\rho)\\
&= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi(\lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor)\\
&= \lfloor((\textbf{let}\ d_1 \Leftarrow \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor.\ \pi_1\ d_1),(\textbf{let}\ d_2 \Leftarrow \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor.\ \pi_2\ d_2))\rfloor\\
&= \lfloor((\pi_1\ (\lfloor 2\rfloor,\lfloor 2\rfloor)),(\pi_2\ (\lfloor 2\rfloor,\lfloor 2\rfloor)))\rfloor\\
&= \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor\\
[\![t_0\ (t_2,t_1)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi([\![(t_2,t_1)]\!]\rho)\\
&= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi(\lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor)\\
&= [\![t_0\ (t_1,t_2)]\!]\rho
\end{aligned}$$

The same result can be obtained by observing that $(t_0\ (t_1,t_2)) = (t_0\ y)[^{(t_1,t_2)}/_y]$ and $(t_0\ (t_2,t_1)) = (t_0\ y)[^{(t_2,t_1)}/_y]$. Then, by compositionality we have:

$$\begin{aligned}
[\![t_0\ (t_1,t_2)]\!]\rho &= \left[\!\!\left[(t_0\ y)[^{(t_1,t_2)}/_y]\right]\!\!\right]\rho\\
&= [\![(t_0\ y)]\!]\rho[^{[\![(t_1,t_2)]\!]\rho}/_y]\\
&= [\![(t_0\ y)]\!]\rho[^{\lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor}/_y]\\
&= [\![(t_0\ y)]\!]\rho[^{[\![(t_2,t_1)]\!]\rho}/_y]\\
&= \left[\!\!\left[(t_0\ y)[^{(t_2,t_1)}/_y]\right]\!\!\right]\rho\\
&= [\![t_0\ (t_2,t_1)]\!]\rho.
\end{aligned}$$

We conclude by showing that the terms $(t_0\ (t_1,t_2))$ and $(t_0\ (t_2,t_1))$ have different canonical forms:

$$\begin{array}{l}
(t_0\ (t_1,t_2)) \to c_1 \quad \nwarrow\ t_0 \to \lambda x'.t, \quad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
\qquad\qquad \nwarrow\ map \to \lambda f'.t', \quad t'[^{\lambda z.\ z}/_{f'}] \to \lambda x'.t, \quad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
\qquad\qquad\quad \nwarrow_{f'=f,t'=\ldots}\ \lambda x.\ (((\lambda z.\ z)\ \textbf{fst}(x)),((\lambda z.\ z)\ \textbf{snd}(x))) \to \lambda x'.t,\\
\qquad\qquad\qquad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
\qquad\qquad\quad \nwarrow_{x'=x,t=\ldots}\ (((\lambda z.\ z)\ \textbf{fst}((t_1,t_2))),((\lambda z.\ z)\ \textbf{snd}((t_1,t_2)))) \to c_1\\
\qquad\qquad \nwarrow_{c_1=(\ldots\ldots)}\ \square\\
(t_0\ (t_2,t_1)) \to c_2 \quad \nwarrow^{*}\ (((\lambda z.\ z)\ \textbf{fst}((t_2,t_1))),((\lambda z.\ z)\ \textbf{snd}((t_2,t_1)))) \to c_2\\
\qquad\qquad \nwarrow_{c_2=(\ldots,\ldots)}\ \square
\end{array}$$

10.11

1. We extend the proof of correctness to take into account the new rules. We recall that the predicate to be proved is

$$P(t \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![t]\!]\rho = [\![c]\!]\rho.$$

For the rule

$$\frac{t \to 0 \quad t_0 \to c_0 \quad t_1 \to c_1}{\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1 \to c_0}$$

we can assume

$$P(t \to 0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![0]\!]\rho = \lfloor 0 \rfloor$$
$$P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$$
$$P(t_1 \to c_1) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1]\!]\rho = [\![c_1]\!]\rho$$

and we want to prove

$$P(\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho = [\![c_0]\!]\rho.$$

We have:

$$\begin{array}{lll} [\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho & = \mathit{Cond}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho) & \text{(by definition)} \\ & = \mathit{Cond}(\lfloor 0 \rfloor, [\![t_0]\!]\rho, [\![t_1]\!]\rho) & \text{(by inductive hypothesis)} \\ & = [\![t_0]\!]\rho & \text{(by definition of } \mathit{Cond}) \\ & = [\![c_0]\!]\rho & \text{(by inductive hypothesis)} \end{array}$$

For the other rule the proof is analogous and thus omitted.

2. As a counterexample, we can take

$$t \stackrel{\text{def}}{=} \textbf{if } 0 \textbf{ then } 1 \textbf{ else rec } x.\ x.$$

In fact, its denotational semantics is

$$[\![t]\!]\rho = \mathit{Cond}([\![0]\!]\rho, [\![1]\!]\rho, [\![\textbf{rec } x.\ x]\!]\rho) = \mathit{Cond}(\lfloor 0 \rfloor, \lfloor 1 \rfloor, \bot_{(V_{int})_\bot} = \lfloor 1 \rfloor$$

and therefore $t \Downarrow$. Vice versa $t \uparrow$, as:

$$\begin{array}{rcl} t \to c & \nwarrow & 0 \to 0, \quad 1 \to c, \quad \textbf{rec } x.\ x \to c' \\ & \nwarrow & 1 \to c, \quad \textbf{rec } x.\ x \to c' \\ & \nwarrow_{c=1} & \textbf{rec } x.\ x \to c' \\ & \nwarrow & x[^{\textbf{rec } x.\ x}/_x] \to c' \\ & = & \textbf{rec } x.\ x \to c' \\ & \nwarrow & \cdots \end{array}$$

10.13 According to the operational semantics we have:

$$\begin{array}{rcl} \textbf{rec } x.\ t \to c & \nwarrow & t[^{\textbf{rec } x.\ t}/_x] \to c \\ & = & t \to c \end{array}$$

because by hypothesis $x \notin \mathrm{fv}(t)$. So we conclude that either both terms have the same canonical form or they do not have any canonical form.

According to the denotational semantics we have

$$\begin{array}{rcl} [\![\textbf{rec } x.\ t]\!]\rho & = & \mathrm{fix}\ \lambda d_x.\ [\![t]\!]\rho[^{d_x}/_x] \\ & = & \mathrm{fix}\ \lambda d_x.\ [\![t]\!]\rho \end{array}$$

When we compute the fixpoint, assuming $t : \tau$, we get:

$$\begin{aligned} d_0 &= \bot_{(V_\tau)_\bot} \\ d_1 &= (\lambda d_x.\ [\![t]\!]\rho)\ d_0 = [\![t]\!]\rho[{}^{d_0}/_{d_x}] = [\![t]\!]\rho \\ d_2 &= (\lambda d_x.\ [\![t]\!]\rho)\ d_1 = [\![t]\!]\rho[{}^{d_1}/_{d_x}] = [\![t]\!]\rho = d_1 \end{aligned}$$

So we have reached the fixpoint and have $[\![\mathbf{rec}\ x.\ t]\!]\rho = [\![t]\!]\rho$.

Alternatively, we could have computed the semantics as follows:

$$\begin{aligned} [\![\mathbf{rec}\ x.\ t]\!]\rho &= [\![t]\!]\rho[{}^{[\![\mathbf{rec}\ x.\ t]\!]\rho}/_x] && \text{(by definition)} \\ &= [\![t[{}^{\mathbf{rec}\ x.\ t}/_x]]\!]\rho && \text{(by Substituition Lemma)} \\ &= [\![t]\!]\rho && \text{(because } x \notin \mathrm{fv}(t)\text{)} \end{aligned}$$

10.14

1. By Theorem 10.1 (Correctness) we have $[\![t_0]\!]\rho = [\![c_0]\!]\rho$. Hence:
$$[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1']\!]\rho[{}^{[\![t_0]\!]\rho}/_x] = [\![t_1']\!]\rho[{}^{[\![c_0]\!]\rho}/_x] = [\![t_1'[{}^{c_0}/_x]]\!]\rho.$$
2. If $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho = \bot_{\mathbb{Z}_\bot}$, then we have that $t_1'[{}^{t_0}/_x] \uparrow$ and $t_1'[{}^{c_0}/_x] \uparrow$, because the operational semantics agrees on convergence with the denotational semantics.
   If $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho \neq \bot_{\mathbb{Z}_\bot}$, it exists $n \in \mathbb{Z}$ such that $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho = \lfloor n \rfloor$. Then, since $t_1'[{}^{t_0}/_x]$ and $t_1'[{}^{c_0}/_x]$ are closed, by Theorem 10.4, we have $t_1'[{}^{t_0}/_x] \to n$ and $t_1'[{}^{c_0}/_x] \to n$.
3. Suppose that $(t_1\ t_0) \to c$ in the eager semantics. Then it must be the case that $t_1 \to \lambda x.\ t_1'$ for some suitable x and t_1', and that $t_1'[c_0/x] \to c$ (we know that $t_0 \to c_0$ by initial hypothesis). Since $c : \mathit{int}$ it must be $c = n$ for some integer n. Then, by the previous point we know that $t_1'[{}^{t_0}/_x] \to n$ because $t_1'[{}^{c_0}/_x] \to n$. We conclude that $(t_1\ t_0) \to c$ in the lazy semantics by exploiting the (lazy) rule for function application.
4. As a simple counterexample, we can take, e.g., $t_1 = \lambda x.\ ((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))$ with $y \notin \mathrm{fv}(t_0)$. In fact, in the lazy semantics, we have:
$$\begin{aligned} (t_1\ t_0) \to c &\quad\nwarrow\ t_1 \to \lambda x'.\ t_1',\quad t_1'[{}^{t_0}/_{x'}] \to c \\ &\quad\nwarrow_{x'=x,\ t_1'=((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))}\ ((\lambda y.\ t_0)\ (\mathbf{rec}\ z.\ z)) \to c \\ &\quad\nwarrow\ (\lambda y.\ t_0) \to \lambda y'.\ t_2,\quad t_2[{}^{(\mathbf{rec}\ z.\ z)}/_{y'}] \to c \\ &\quad\nwarrow_{y'=y,\ t_2=t_0}\ t_0[{}^{(\mathbf{rec}\ z.\ z)}/_y] \to c \\ &\quad= t_0 \to c \\ &\quad\nwarrow_{c=c_0}\ \square \end{aligned}$$

   Whereas in the eager semantics we have:

$$
\begin{array}{rl}
(t_1\ t_0) \to c & \nwarrow\ t_1 \to \lambda x'.\, t_1',\quad t_0 \to c',\quad t_1'[{}^{c'}/_{x'}] \to c \\
\nwarrow_{x'=x,\ t_1'=((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))} & t_0 \to c',\quad ((\lambda y.\ c')\ (\mathbf{rec}\ z.\ z)) \to c \\
\nwarrow_{c'=c_0} & ((\lambda y.\ c_0)\ (\mathbf{rec}\ z.\ z)) \to c \\
\nwarrow & (\lambda y.\ c_0) \to \lambda y'.\, t_2,\quad \mathbf{rec}\ z.\ z \to c'',\quad t_2[{}^{c''}/_{y'}] \to c \\
\nwarrow_{y'=y,\ t_2=t_0} & \mathbf{rec}\ z.\ z \to c'',\quad t_0[{}^{c''}/_{y}] \to c \\
\nwarrow & \mathbf{rec}\ z.\ z \to c'',\quad t_0[{}^{c''}/_{y}] \to c \\
\nwarrow & \cdots
\end{array}
$$

5. Let $x, t_0 : \tau_0$ and $y : \tau$ and assume $t_0 \neq c_0$. As a last counterexample, let us take $t_1' = \lambda y.\ x$ (and $t_1 = \lambda x.\ t_1'$), with $t_1' : \tau \to \tau_0$. We have immediately that $t_1'[{}^{t_0}/_{x}] = \lambda y.\ t_0$ and $t_1'[{}^{c_0}/_{x}] = \lambda y.\ c_0$ are already in canonical form and they are different. Moreover, $(t_1\ t_0) \to \lambda y.\ t_0$ in the lazy semantics, but $(t_1\ t_0) \to \lambda y.\ c_0$ in the eager semantics.

Problems of Chapter 11

11.3 Let us take the relation

$$R \stackrel{\text{def}}{=} \{(B_k^n, B_{i_1}^1 \mid \cdots \mid B_{i_n}^1)\ \mid\ \sum_{j=1}^{n} i_j = k\}$$

We show that R is a strong bisimulation.

- If $k = 0$ we have that $(B_0^n, B_{i_1}^1 \mid \cdots \mid B_{i_n}^1) \in R$ iff $\forall j \in [1,n].\ i_j = 0$. Then there is a unique transitions leaving B_0^n, namely $B_0^n \xrightarrow{in} B_1^n$, while we have n different transitions leaving $B_0^1 \mid \cdots \mid B_0^1$, one for each buffer. The states reached are all those processes $B_{i_1}^1 \mid \cdots \mid B_{i_n}^1$ such that $\sum_{j=1}^n i_j = 1$. In fact, we have that B_1^n is related via R to any such process.
- If $k = n$ we have that $(B_n^n, B_{i_1}^1 \mid \cdots \mid B_{i_n}^1) \in R$ iff $\forall j \in [1,n].\ i_j = 1$. Then there is a unique transitions leaving B_n^n, namely $B_n^n \xrightarrow{out} B_{n-1}^n$, while we have n different transitions leaving $B_1^1 \mid \cdots \mid B_1^1$, one for each buffer. The states reached are all those processes $B_{i_1}^1 \mid \cdots \mid B_{i_n}^1$ such that $\sum_{j=1}^n i_j = n-1$. In fact, we have that B_{n-1}^n is related via R to any such process.
- If $0 < k < n$ we have that $(B_k^n, B_{i_1}^1 \mid \cdots \mid B_{i_n}^1) \in R$ iff $\sum_{j=1}^n i_j = k$. Then there are two transitions leaving B_k^n, namely $B_k^n \xrightarrow{in} B_{k+1}^n$ and $B_k^n \xrightarrow{out} B_{k-1}^n$, while we have $n-k$ different *in*-transitions leaving $B_{i_1}^1 \mid \cdots \mid B_{i_n}^1$, one for each empty buffer and k different *out*-transitions, one for each full buffer. In the first case, the states reached are all those processes $B_{i_1}^1 \mid \cdots \mid B_{i_n}^1$ such that $\sum_{j=1}^n i_j = k+1$, because one empty buffer has become full, and we have that B_{k+1}^n is related by R to any such process. In the second case, the states reached are all those processes $B_{i_1}^1 \mid \cdots \mid B_{i_n}^1$ such that $\sum_{j=1}^n i_j = k-1$, because one of the full buffers has become empty, and we have that B_{k-1}^n is related by R to any such process.

11.6

Conditionals: We encode the conditional statement by testing in input the value stored in x and setting the continuation to p_1 only when such value is i (in all the other cases, the continuation is p_2):

$$\begin{array}{l}\overline{xr_1}.p_2 \;+\; ... \;+\; \overline{xr_{i-1}}.p_2 \;+\\ \overline{xr_i}.p_1 \;+\\ \overline{xr_{i+1}}.p_2 \;+\; ... \;+\; \overline{xr_n}.p_2\end{array}$$

Iteration: Let ϕ be the permutation that switches d and *done*. By using the recursive process, we let

$$\begin{array}{l}\textbf{rec}\ W.\ \ \overline{xr_1}.Done \;+\; ... \;+\; \overline{xr_{i-1}}.Done+\\ \qquad\quad \overline{xr_i}.(p[\phi] \mid d.W)\backslash d \;+\\ \qquad\quad \overline{xr_{i+1}}.Done \;+\; ... \;+\; \overline{xr_n}.Done\end{array}$$

that, in the case the value i can be read from x, activates the continuation

$$(p[\phi] \mid d.\textbf{rec}\ W.\ (...))\backslash d$$

that executes p and (if and) when it terminates activates another instance of the recursive process. In all the other cases it activates the termination process *Done*.

Concurrency: Let ϕ_i be the permutation that switches d_i and *done*. We encode the concurrent execution of c_1 and c_2 as

$$(\ p_1[\phi_1] \mid (p_2[\phi_2]) \mid d_1.d_2.Done\ )\backslash d_1\backslash d_2$$

Note that we can use the simpler process

$$d_1.d_2.Done$$

to wait for the termination of $p_1[\phi_1]$ and $p_2[\phi_2]$ instead of the more complex process

$$d_1.d_2.Done + d_2.d_1.Done$$

because the termination message cannot be released anyway until both p_1 and p_2 have terminated.

11.7 We show that strong bisimilarity is a congruence w.r.t. sum. Formally, we want to prove that for any CCS processes p_1, p_2, q_1, q_2 we have that

$$p_1 \simeq q_1 \wedge p_2 \simeq q_2 \quad \text{implies} \quad p_1 + p_2 \simeq q_1 + q_2$$

Let us assume the premise $p_1 \simeq q_1 \wedge p_2 \simeq q_2$; we want to prove that $p_1 + p_2 \simeq q_1 + q_2$. Since $p_1 \simeq q_1$, there exists a strong bisimulation R_1 such that $p_1\ R_1\ q_1$. Since

$p_2 \simeq q_2$, there exists a strong bisimulation R_2 such that $p_2\ R_2\ q_2$. We want to find a relation R such that:

1. $p_1 + p_2\ R\ q_1 + q_2$
2. R is a strong bisimulation (i.e., $R \subseteq \Phi(R)$;

Let us define R as follows and then prove that it is a strong bisimulation:

$$R \stackrel{\text{def}}{=} \{(p_1 + p_2, q_1 + q_2)\} \cup R_1 \cup R_2$$

Obviously, we have $p_1 + p_2\ R\ q_1 + q_2$, by definition of R. For the second point, we need to take a pair in R and prove that it satisfies the definition of bisimulation. Let us consider the various cases.

- for the pairs in R_1 and R_2 the proof is trivial, since R_1 and R_2 are bisimulation themselves and they are included in R.
- Take $(p_1 + p_2, q_1 + q_2) \in R$ and take μ, p such that $p_1 + p_2 \xrightarrow{\mu} p$. We want to prove that there exists q with $q_1 + q_2 \xrightarrow{\mu} q$ and $p\ R\ q$. Since $p_1 + p_2 \xrightarrow{\mu} p$, by the operational semantics of CCS it must be the case that either $p_1 \xrightarrow{\mu} p$ or $p_2 \xrightarrow{\mu} p$.
  - If $p_1 \xrightarrow{\mu} p$, since $p_1\ R_1\ q_1$, there exists q with $q_1 \xrightarrow{\mu} q$ and $p\ R_1\ q$. Then $q_1 + q_2 \xrightarrow{\mu} q$ and $(p,q) \in R_1 \subseteq R$, so we are done.
  - If $p_2 \xrightarrow{\mu} p$, since $p_2\ R_2\ q_2$, there exists q with $q_2 \xrightarrow{\mu} q$ and $p\ R_2\ q$. Then $q_1 + q_2 \xrightarrow{\mu} q$ and $(p,q) \in R_2 \subseteq R$, so we are done.

  The case where $p_1 + p_2$ has to (bi)simulate a transition $q_1 + q_2 \xrightarrow{\mu} q$ is analogous to the previous case.

11.15

1. Suppose R is a loose bisimulation. We want to show that it is a weak bisimulation. Take any pair $(p,q) \in R$ and any transition $p \xrightarrow{\mu} p'$. We want to prove that there exists some q' such that $q \stackrel{\mu}{\Rightarrow} q'$ and $(p',q') \in R$. By definition of $\stackrel{\mu}{\Rightarrow}$ we have $p \stackrel{\mu}{\Rightarrow} p'$. Since R is a loose bisimulation, there must exist some q' such that $q \stackrel{\mu}{\Rightarrow} q'$ with $(p',q') \in R$ and we conclude. The case when p has to (bi)simulate a transition of q is analogous and thus omitted.
2. Suppose R is a weak bisimulation. We want to show that it is a loose bisimulation. Take any pair $(p,q) \in R$ and any weak transition $p \stackrel{\mu}{\Rightarrow} p'$. (The case when $q \stackrel{\mu}{\Rightarrow} q'$ is analogous and thus omitted). We want to prove that there exists q' such that $q \stackrel{\mu}{\Rightarrow} q'$ and $(p',q') \in R$. We first prove by mathematical induction on n that if

   $$p \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_n$$

   then there exists some q' such that $q \stackrel{\tau}{\Rightarrow} q'$ and $(p_n,q') \in R$.

   - The base case is when $n = 0$, i.e., $p_n = p$. Then we just take $q' = q$.

- for the inductive case, suppose the property holds for n, we want to prove it for $n+1$. Suppose
$$p \xrightarrow{\tau} p_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_n \xrightarrow{\tau} p_{n+1}.$$
By inductive hypothesis, there exists q'' such that $q \stackrel{\tau}{\Rightarrow} q''$ and $(p_n, q'') \in R$. Since R is a weak bisimulation and $p_n \xrightarrow{\tau} p_{n+1}$, there exists some q' such that $q'' \stackrel{\tau}{\Rightarrow} q'$ and $(p_{n+1}, q') \in R$. Since $q \stackrel{\tau}{\Rightarrow} q'' \stackrel{\tau}{\Rightarrow} q'$ we have $q \stackrel{\tau}{\Rightarrow} q'$ and we are done.

Now we distinguish two cases:

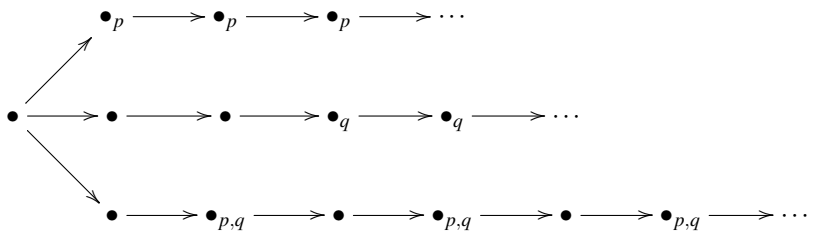
- If $\mu = \tau$, since $p \stackrel{\tau}{\Rightarrow} p'$, there exist $p_1, ..., p_n$ such that
$$p \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_n = p'$$
and, by the argument above, there exists q' such that $q \stackrel{\tau}{\Rightarrow} q'$ and $(p', q') \in R$.
- If $\mu \neq \tau$, since $p \stackrel{\mu}{\Rightarrow} p'$, there exist p'', p''' such that
$$p \stackrel{\tau}{\Rightarrow} p'' \xrightarrow{\mu} p''' \stackrel{\tau}{\Rightarrow} p'.$$
By the argument above, we can find q'' such that $q \stackrel{\tau}{\Rightarrow} q''$ and $(p'', q'') \in R$. Since $p'' \xrightarrow{\mu} p'''$ and R is a weak bisimulation, there exists q''' such that $q'' \stackrel{\mu}{\Rightarrow} q'''$ and $(p''', q''') \in R$. By the argument above, we can find q' such that $q''' \stackrel{\tau}{\Rightarrow} q'$ and $(p', q') \in R$. Since $q \stackrel{\tau}{\Rightarrow} q'' \stackrel{\mu}{\Rightarrow} q''' \stackrel{\tau}{\Rightarrow} q'$ we have $q \stackrel{\mu}{\Rightarrow} q'$ and we are done.

Problems of Chapter 12

12.1

1. Mutual exclusion: $G\ \neg(use_1 \wedge use_2)$.
2. Release: $G\ (use_i \Rightarrow F\ rel_i)$.
3. Priority: $G\ ((req_1 \wedge req_2) \Rightarrow ((\neg use_2)\ U\ (use_1 \wedge \neg use_2)))$.
4. Absence of starvation: $G\ (req_i \Rightarrow F\ use_i)$

12.3 The CTL* formula $\phi \stackrel{\text{def}}{=} AF\ G\ (p \vee O\ q)$ expresses the property that along all paths we can enter a state v such that, from that moment on, any state that does not satisfy p is followed by a state that satisfies q. A simple branching structure where ϕ is satisfied is:



The formula ϕ is an LTL formula (as LTL formulas are tacitly quantified by the path operator A), but it is not a CTL formula (because the linear operators G and O are not preceded by path operators).

12.6 We let $\phi \stackrel{\text{def}}{=} \nu x.\ (p \vee \Diamond x) \wedge (q \vee \Box x)$. We have

$$\begin{aligned}
[\![\nu x.\ (p \vee \Diamond x) \wedge (q \vee \Box x)]\!]\rho &\stackrel{\text{def}}{=} \text{FIX } \lambda S.\ [\![(p \vee \Diamond x) \wedge (q \vee \Box x)]\!]\rho[^S/_x] \\
&= \text{FIX } \lambda S.\ [\![p \vee \Diamond x]\!]\rho[^S/_x] \cap [\![q \vee \Box x]\!]\rho[^S/_x] \\
&= \text{FIX } \lambda S.\ ([\![p]\!]\rho[^S/_x] \cup [\![\Diamond x]\!]\rho[^S/_x]) \cap \\
&\qquad\qquad ([\![q]\!]\rho[^S/_x] \cup [\![\Box x]\!]\rho[^S/_x]) \\
&= \text{FIX } \lambda S.\ (\rho(p) \cup \{v \mid \exists v' \in S. v \to v'\}) \cap \\
&\qquad\qquad (\rho(q) \cup \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S\})
\end{aligned}$$

Let $V = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. We have $\rho(p) = \{s_6\}$ and $\rho(q) = \{s_3\}$ We compute the fixpoint by successive approximations:

$$\begin{aligned}
S_0 &= V \\
S_1 &= (\{s_6\} \cup \{v \mid \exists v' \in V. v \to v'\}) \cap (\{s_3\} \cup \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in V\}) \\
&= (\{s_6\} \cup \{s_1, s_2, s_4, s_5\}) \cap (\{s_3\} \cup V) \\
&= \{s_1, s_2, s_4, s_5, s_6\} \\
S_2 &= (\{s_6\} \cup \{v \mid \exists v' \in S_1. v \to v'\}) \cap (\{s_3\} \cup \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_1\}) \\
&= (\{s_6\} \cup \{s_1, s_2, s_4, s_5\}) \cap (\{s_3\} \cup \{s_1, s_3, s_4, s_5, s_6\}) \\
&= \{s_1, s_4, s_5, s_6\} \\
S_3 &= (\{s_6\} \cup \{v \mid \exists v' \in S_2. v \to v'\}) \cap (\{s_3\} \cup \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_2\}) \\
&= (\{s_6\} \cup \{s_1, s_2, s_4, s_5\}) \cap (\{s_3\} \cup \{s_3, s_4, s_5, s_6\}) \\
&= \{s_4, s_5, s_6\} \\
S_4 &= (\{s_6\} \cup \{v \mid \exists v' \in S_3. v \to v'\}) \cap (\{s_3\} \cup \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_3\}) \\
&= (\{s_6\} \cup \{s_1, s_2, s_4, s_5\}) \cap (\{s_3\} \cup \{s_3, s_4, s_5, s_6\}) \\
&= \{s_4, s_5, s_6\} = S_3
\end{aligned}$$

We have reached the (greatest) fixpoint and therefore $[\![\phi]\!]\rho = \{s_4, s_5, s_6\}$.

12.8 We let $\phi \stackrel{\text{def}}{=} \nu x.\ (p \wedge \Box x)$. We have

$$\begin{aligned}[\![\nu x.\ (p \wedge \Box x)]\!]\rho &\stackrel{\text{def}}{=} \text{FIX } \lambda S.\ [\![p \wedge \Box x]\!]\rho[^S/_x] \\ &= \text{FIX } \lambda S.\ [\![p]\!]\rho[^S/_x] \cap [\![\Box x]\!]\rho[^S/_x] \\ &= \text{FIX } \lambda S.\ \rho(p) \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S\}\end{aligned}$$

Let $V = \{s_1, s_2, s_3, s_4\}$. We have $\rho(p) = \{s_1, s_3, s_4\}$ We compute the fixpoint by successive approximations:

$$\begin{aligned}S_0 &= V \\ S_1 &= \{s_1, s_3, s_4\} \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in V\} \\ &= \{s_1, s_3, s_4\} \cap \{s_1, s_2, s_3, s_4\} \\ &= \{s_1, s_3, s_4\} \\ S_2 &= \{s_1, s_3, s_4\} \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_1\} \\ &= \{s_1, s_3, s_4\} \cap \{s_2, s_3, s_4\} \\ &= \{s_3, s_4\} \\ S_3 &= \{s_1, s_3, s_4\} \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_2\} \\ &= \{s_1, s_3, s_4\} \cap \{s_2, s_4\} \\ &= \{s_4\} \\ S_4 &= \{s_1, s_3, s_4\} \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S_3\} \\ &= \{s_1, s_3, s_4\} \cap \{s_4\} \\ &= \{s_4\} = S_3\end{aligned}$$

We have reached the (greatest) fixpoint and therefore $[\![\phi]\!]\rho = \{s_4\}$.

Problems of Chapter 13

13.1 Let us consider processes such that whenever they contain an output prefixed sub-term $\bar{x}y.p$ then $p = \mathbf{nil}$. We abbreviate $\bar{x}y.\mathbf{nil}$ as $\bar{x}y$. Let us try to encode ordinary (synchronous) π-calculus in asynchronous π-calculus. It is instructive to proceed by successive attempts.

1. Let us define a first simple mapping $\mathscr{A}$ from synchronous processes to asynchronous ones. The mapping is the identity except for output prefixed processes, that have no correspondence in asynchronous π-calculus. Thus we let $\mathscr{A}$ be (the homomorphic extension of) the function such that:

   $$\mathscr{A}(\bar{x}y.p) \stackrel{\text{def}}{=} \bar{x}y \mid \mathscr{A}(p)$$

   Unfortunately, this solution is not satisfactory, because (the translated version of) p can be executed before the message $\bar{x}y$ gets received.

2. As a second attempt, we can try to prefix (the translated version of) p by the input of an acknowledgment over some fixed channel a. Of course, we must then revise also the translation of input prefixes, so to send the acknowledgment. We let:

$$\begin{aligned} \mathscr{A}(\overline{x}y.p) &\stackrel{\text{def}}{=} \overline{x}y \mid a(x_k).\mathscr{A}(p) \qquad \text{with } x_k \notin \text{fn}(p) \\ \mathscr{A}(x(z).q) &\stackrel{\text{def}}{=} x(z).(\overline{a}a \mid \mathscr{A}(q)) \end{aligned}$$

   Also this solution has some pitfall, because if a unique channel a is used to send all the acknowledgments, then communications can interfere one with each other.
3. As a third attempt, we enforce the sharing of a private channel a for sending the acknowledgment. The sender creates the channel and transmit it to the receiver, the receiver gets the private channel and uses it to receive the message, then uses it to send the acknowledgment. Consequently, we let

$$\begin{aligned} \mathscr{A}(\overline{x}y.p) &\stackrel{\text{def}}{=} (a)(\overline{x}a \mid \overline{a}y \mid a(x_k).\mathscr{A}(p)) \qquad \text{with } a, x_k \notin \text{fn}(\overline{x}y.p) \\ \mathscr{A}(x(z).q) &\stackrel{\text{def}}{=} x(x_a).a(z)(\overline{x_a}x_a \mid \mathscr{A}(q)) \qquad \text{with } x_a \notin \text{fn}(x(z).q) \end{aligned}$$

   But then it is immediate to spot that, in the encoding of the sender, the message $\overline{a}y$ can be directly taken in input from the input prefix $a(x_k)$ that is running in parallel, waiting for the acknowledgment.
4. As a fourth attempt, we introduce two different private channels: one for receiving the acknowledgment and one for sending the data:

$$\begin{aligned} \mathscr{A}(\overline{x}y.p) &\stackrel{\text{def}}{=} (a)(\overline{x}a \mid a(x_k).(\overline{x_k}y \mid \mathscr{A}(p))) \qquad \text{with } a, x_k \notin \text{fn}(\overline{x}y.p) \\ \mathscr{A}(x(z).q) &\stackrel{\text{def}}{=} x(x_a).(k)(\overline{x_a}k \mid k(z).\mathscr{A}(q)) \qquad \text{with } x_a, k \notin \text{fn}(x(z).q) \end{aligned}$$

   This solution works fine: $\mathscr{A}(p)$ can be executed only after a receiver has started the interaction protocol and has sent a message on a; vice versa, $\mathscr{A}(q)$ can be executed only after the actual message has been received. However the above solution requires three asynchronous communications to implement a single synchronous communication.
5. As a fifth attempt, we try to improve the efficiency of the fourth solution by switching the role of the sender and the receiver in starting the protocol: it is the receiver that sends the first message on x, expressing its intention to receive some data. The sender waits for some receiver to start the interaction and then sends the data.

$$\begin{aligned} \mathscr{A}(\overline{x}y.p) &\stackrel{\text{def}}{=} x(x_a).(\overline{x_a}y \mid \mathscr{A}(p)) \qquad \text{with } x_a \notin \text{fn}(\overline{x}y.p) \\ \mathscr{A}(x(z).q) &\stackrel{\text{def}}{=} (a)(\overline{x}a \mid a(z).\mathscr{A}(q)) \qquad \text{with } a \notin \text{fn}(x(z).q) \end{aligned}$$

   Nicely, this solution only requires two asynchronous communications to implement a single synchronous communication.

13.2 The polyadic π-calculus allows prefixes of the form

$$\pi \ ::= \ \tau \ | \ x(z_1,...,z_n) \ | \ \overline{x}\langle y_1,...,y_n\rangle$$

Monadic processes are just the particular instance of polyadic ones (when $n = 1$ for all prefixes).

Suppose we have defined a type system that guarantees that any two input and output prefixes that may occur on the same channel carry the same number of arguments. We restrict to encode only such well-typed process. Let us try to encode polyadic π-calculus in ordinary (monadic) π-calculus. As for Problem 13.1, it is instructive to proceed by successive attempts.

1. Let us define a first simple mapping $\mathscr{M}$ from polyadic processes to monadic ones. The mapping is the identity except for input and output prefixed processes. Thus we let $\mathscr{M}$ be (the homomorphic extension of) the function such that:

$$\mathscr{M}(\overline{x}\langle y_1,...,y_n\rangle.p) \stackrel{\text{def}}{=} \overline{x}y_1.\cdots.\overline{x}y_n.\mathscr{M}(p))$$
$$\mathscr{M}(x(z_1,...,z_n).q) \stackrel{\text{def}}{=} x(z_1).\cdots.x(z_n).\mathscr{M}(q))$$

   Unfortunately, this solution is not satisfactory, because if there are many senders and receivers on the same channel x that run in parallel, then their sequence of interactions can be mixed.
2. As a second attempt, we consider the possibility to exchange a private name in the first communication and then to use this private name to send the sequence of arguments. We modify the definition of $\mathscr{M}$ accordingly:

$$\mathscr{M}(\overline{x}\langle y_1,...,y_n\rangle.p) \stackrel{\text{def}}{=} (c)\overline{x}c.\overline{c}y_1.\cdots.\overline{c}y_n.\mathscr{M}(p)$$
$$\mathscr{M}(x(z_1,...,z_n).q) \stackrel{\text{def}}{=} x(x_c).x_c(z_1).\cdots.x_c(z_n).\mathscr{M}(q)$$

   with $c \notin \text{fn}(\overline{x}\langle y_1,...,y_n\rangle.p)$ and $x_c \notin \text{fn}(q) \cup \{z_1,...,z_n\}$.

13.3 Let us consider the following syntax for HOπ, the Higher Order π-calculus:

$$P \ ::= \ \mathbf{nil} \ | \ \pi.P \ | \ P|Q \ | \ (y)P \ | \ Y$$
$$\pi \ ::= \ \tau \ | \ x(y) \ | \ \overline{x}y \ | \ x(Y) \ | \ \overline{x}\langle P\rangle$$

where x, y are names and X, Y are process variables. The process output prefix $\overline{x}\langle P\rangle$ can be use to send a process P on the channel x, while the process input prefix $x(Y)$ can be used to receive the process P and to assign it to the process variable Y. Without delving into the detail of operational and abstract semantics for HOπ, higher order communication can be realised by transitions such as:

$$\overline{x}\langle P\rangle.Q \mid x(Y).R \stackrel{\tau}{\rightarrow} Q \mid R[^P/_Y]$$

For example, replication $!P$ can be coded in HOπ by the process:

$$(r)(\ Dup \mid \overline{r}\langle P \mid Dup\rangle.\mathbf{nil}\ )$$

where $Dup \stackrel{\text{def}}{=} r(X).(X \mid \overline{r}\langle X\rangle.\mathbf{nil})$. Roughly, process *Dup* waits to receive a process on r, stores it in X, spawns a copy of X and re-sends X on r. When *Dup* runs in parallel with $\overline{r}\langle P \mid Dup\rangle.\mathbf{nil}$, then the process $P \mid Dup$ is released together with a further activation $\overline{r}\langle P \mid Dup\rangle.\mathbf{nil}$, so that many more copies of P can be created in the same way. The name r is restricted so to avoid interferences from the environment.

To encode HOπ in (ordinary, monadic) π-calculus, the idea is to encode a process output prefix $\overline{x}\langle P\rangle$ by installing a server that spawns new copies of P upon requests on a (private) channel p which is communicated in place of P. Then, the name passing mechanism of π-calculus allows to encode process input prefixes like $x(Y)$ as ordinary input prefixes $x(x_Y)$ that will receive the name p and bind it to the variable x_Y, which, in turn, can be used to invoke the server associated with P by replacing all occurrences of Y with the simple process $\overline{x_Y}x_Y.\mathbf{nil}$ (like a service invocation). Formally, we define a mapping $\mathscr{H}$ from HOπ processes to π-calculus ones as the homomorphic extension of the function such that:

$$\begin{aligned}
\mathscr{H}(\overline{x}\langle P\rangle.Q) &\stackrel{\text{def}}{=} (p)(\,\mathscr{H}(Q) \mid !p(x_p).\mathscr{H}(P)\,) \qquad \text{with } p,x_p \notin \text{fn}(P)\\
\mathscr{H}(x(Y).R) &\stackrel{\text{def}}{=} x(x_Y).\mathscr{H}(R) \qquad \text{with } x_Y \notin \text{fn}(R)\\
\mathscr{H}(Y) &\stackrel{\text{def}}{=} \overline{x_Y}x_Y.\mathbf{nil}
\end{aligned}$$

where we assume a reserved set of names x_Y is available, one for each process variable Y.

13.4 By using the axioms for structural congruence, we have (with $x \notin \text{fn}(p)$):

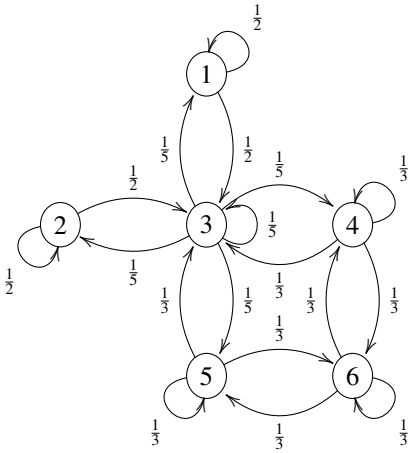
$$(x)p \equiv (x)(p \mid \mathbf{nil}) \equiv p \mid (x)\,\mathbf{nil} \equiv p \mid \mathbf{nil} \equiv p$$

13.5 Take $p \stackrel{\text{def}}{=} \mathbf{nil}$ and $q \stackrel{\text{def}}{=} \mathbf{nil} \mid (x)\overline{x}y.\mathbf{nil}$. We have $\text{fn}(p) = \varnothing$ and $\text{fn}(q) = \{y\}$, but both p and q have no outgoing transitions and therefore are strong early full bisimilar.

Problems of Chapter 14

14.2

1. The PTS and its transition matrix P are



$$P = \begin{vmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{vmatrix}$$

2. It is immediate to check that the DTMC is ergodic, as it it strongly connected and has self-loops. We compute the steady state distribution. The corresponding system of linear equations is

$$\begin{cases} \frac{1}{2}\pi_1 + \frac{1}{5}\pi_3 = \pi_1 \\ \frac{1}{2}\pi_2 + \frac{1}{5}\pi_3 = \pi_2 \\ \frac{1}{2}\pi_1 + \frac{1}{2}\pi_2 + \frac{1}{5}\pi_3 + \frac{1}{3}\pi_4 + \frac{1}{3}\pi_5 = \pi_3 \\ \frac{1}{5}\pi_3 + \frac{1}{3}\pi_4 + \frac{1}{3}\pi_6 = \pi_4 \\ \frac{1}{5}\pi_3 + \frac{1}{3}\pi_5 + \frac{1}{3}\pi_6 = \pi_5 \\ \frac{1}{3}\pi_4 + \frac{1}{3}\pi_5 + \frac{1}{3}\pi_6 = \pi_6 \\ \pi_1 + \pi_2 + \pi_3 + \pi_4 + \pi_5 + \pi_6 = 1 \end{cases}$$

from which we derive

$$\begin{cases} \pi_1 = \frac{2}{5}\pi_3 \\ \pi_2 = \frac{2}{5}\pi_3 \\ \frac{2}{5}\pi_3 = \frac{1}{3}(\pi_4 + \pi_5) \\ \pi_6 = \frac{3}{5}\pi_3 \\ \pi_4 = \frac{3}{5}\pi_3 \\ \pi_5 = \frac{3}{5}\pi_3 \\ \frac{18}{5}\pi_3 = 1. \end{cases}$$

Therefore: $\pi_3 = \frac{5}{18}$, $\pi_1 = \pi_2 = \frac{2}{18}$ and $\pi_3 = \pi_4 = \pi_5 = \frac{3}{18}$, i.e., $\pi = \left| \frac{2}{18} \ \frac{2}{18} \ \frac{5}{18} \ \frac{3}{18} \ \frac{3}{18} \ \frac{3}{18} \right|$.

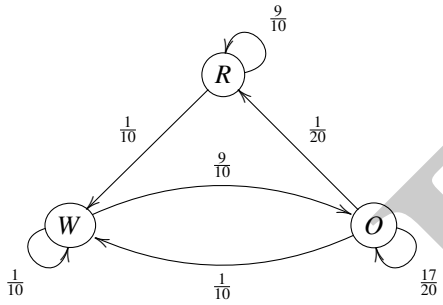
3. We have:

$$\pi^{(0)} = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$
$$\pi^{(1)} = \pi^{(0)}P = \begin{vmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \end{vmatrix}$$
$$\pi^{(2)} = \pi^{(1)}P = \begin{vmatrix} \frac{1}{4}+\frac{1}{10} & \frac{1}{10} & \frac{1}{4}+\frac{1}{10} & \frac{1}{10} & \frac{1}{10} & 0 \end{vmatrix} = \begin{vmatrix} \frac{7}{20} & \frac{1}{10} & \frac{7}{20} & \frac{1}{10} & \frac{1}{10} & 0 \end{vmatrix}$$
$$\pi^{(3)} = \pi^{(2)}P = \begin{vmatrix} \ldots & \frac{1}{30}+\frac{1}{30} \end{vmatrix} = \begin{vmatrix} \ldots & \frac{1}{15} \end{vmatrix}.$$

Thus, the probability to find the mouse in room 6 after three steps is $\frac{1}{15}$.

14.6

1. The PTS and its transition matrix P (with states ordered as R, W, O) are



$$P = \begin{vmatrix} \frac{9}{10} & \frac{1}{10} & 0 \\ 0 & \frac{1}{10} & \frac{9}{10} \\ \frac{1}{20} & \frac{1}{10} & \frac{17}{20} \end{vmatrix}$$

   It is immediate to check that the DTMC is ergodic, as it it strongly connected and has self-loops.
2. Since the machine is waiting at time t, we can assume $\pi^{(t)} = \begin{vmatrix} 0 & 1 & 0 \end{vmatrix}$. Then, $\pi^{(t+1)} = \pi^{(t)}P = \begin{vmatrix} 0 & \frac{1}{10} & \frac{9}{10} \end{vmatrix}$ and the probability to be operating is 0.9.
3. We compute the steady state distribution. The corresponding system of linear equations is

$$\begin{cases} \frac{9}{10}\pi_1 + \frac{1}{20}\pi_3 = \pi_1 \\ \frac{1}{10}\pi_1 + \frac{1}{10}\pi_2 + \frac{1}{10}\pi_3 = \pi_2 \\ \frac{9}{10}\pi_2 + \frac{17}{20}\pi_3 = \pi_3 \\ \pi_1 + \pi_2 + \pi_3 = 1 \end{cases}$$

   from which we derive $\pi_1 = \frac{3}{10}$, $\pi_2 = \frac{1}{10}$ and $\pi_3 = \frac{3}{5}$ and the probability to be operating on the long run is 0.6.

14.8

1. The embedded DTMC is defined by the matrix

$$P = \begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix}.$$

2. Let $\pi^{(0)} = \begin{vmatrix} p & q \end{vmatrix}$ for some $p, q \in [0.1]$ with $q = 1 - p$. We have $\pi^{(1)} = \pi^{(0)}P = \begin{vmatrix} q & p \end{vmatrix}$ and $\pi^{(2)} = \pi^{(1)}P = \begin{vmatrix} p & q \end{vmatrix} = \pi^{(0)}$. Therefore:

$$\pi^{(t)} = \left| p^{((t+1) \bmod 2)} q^{(t \bmod 2)} \quad p^{(t \bmod 2)} q^{((t+1) \bmod 2)} \right|.$$

Note that the embedded DTMC exhibits a periodic behaviour that does not depend in any way from the rates λ and μ.

14.10

1. The sojourn time probability is defined as the probability of not leaving the state. We have:
$$P(X_t = s_0 \mid X_0 = s_0) = e^{-\lambda t}$$
with $\lambda = \lambda_1 + \lambda_1 = 2\lambda_1$.
2. Since the sum of all the rates for the transitions leaving s_0 is $2\lambda_1$ and s_1 and s_3 have one single outgoing transition each with rate $\lambda_2 > 2\lambda_1$, then s_0 cannot be equivalent to s_1 and s_3. For the same reason, s_2 is not equivalent to s_1 and s_3. Then, let us consider the equivalence relation $R \stackrel{\text{def}}{=} \{ \{s_0, s_2\}, \{s_1, s_3\} \}$. Next, we show that R is a bisimulation relation. Let $I_1 = \{s_0, s_2\}$ and $I_2 = \{s_1, s_3\}$. We have:

$$\begin{array}{ll} \gamma_C(s_0, I_1) = \lambda_1 & \gamma_C(s_0, I_2) = \lambda_1 \\ \gamma_C(s_2, I_1) = \lambda_1 & \gamma_C(s_2, I_2) = \lambda_1 \\ \gamma_C(s_1, I_1) = \lambda_2 & \gamma_C(s_1, I_2) = 0 \\ \gamma_C(s_3, I_1) = \lambda_2 & \gamma_C(s_3, I_2) = 0. \end{array}$$

Problems of Chapter 15

15.1 It can be seen that reactive, generative and simple Segala models can all be expressed in terms of Segala models.

- Take a reactive model $\alpha_{\mathsf{r}} : S \to L \to (\mathscr{D}(S) \cup 1)$. We can define its corresponding Segala model $\alpha_s : S \to \wp(\mathscr{D}(L \times S))$ as follows, for any $s \in S, \ell \in L$:
  - if $\alpha_{\mathsf{r}}(s)(\ell) = *$, then α_s must be such that $\forall d \in \alpha_s(s)$ and $\forall s' \in S$ it holds $d(\ell, s') = 0$;
  - if $\alpha_{\mathsf{r}}(s)(\ell) = d$ (with $d \in \mathscr{D}(S)$), then there is $d_\ell \in \alpha_s(s)$ such that $\forall \ell' \in L$, $\ell' \neq \ell$ and $\forall s' \in S$ it holds $d_\ell(\ell', s') = 0$ and $d_\ell(\ell, s') = d(s')$.
- Take a generative model $\alpha_{\mathsf{g}} : S \to (\mathscr{D}(L \times S) \cup 1)$. We can define its corresponding Segala model $\alpha_s : S \to \wp(\mathscr{D}(L \times S))$ as follows, for any $s \in S$:
  - if $\alpha_{\mathsf{g}}(s) = *$, then we simply set $\alpha_s(s) = \varnothing$;
  - if $\alpha_{\mathsf{g}}(s) = d$ (with $d \in \mathscr{D}(L \times S)$), then we let $\alpha_s(s) = \{d\}$.
- Take a simple Segala model $\alpha_{\mathsf{sim}} : S \to \wp(L \times \mathscr{D}(S))$. We can define its corresponding Segala model $\alpha_s : S \to \wp(\mathscr{D}(L \times S))$ as follows, for any $s \in S$:
  - if $(\ell, d) \in \alpha_{\mathsf{sim}}(s)$ (with $d \in \mathscr{D}(S)$), then there is $d_{(\ell,d)} \in \alpha_s(s)$ such that $\forall \ell' \in L$, $\ell' \neq \ell$ and $\forall s' \in S$ it holds $d_{(\ell,d)}(\ell', s') = 0$ and $d_{(\ell,d)}(\ell, s') = d(s')$.

Note that representing generative models in simple Segala ones is not always possible. This is due to the fact that, in general, when $\alpha_{\mathsf{g}}(s) = d \in \mathscr{D}(L \times S)$ then d can assigns probabilities to pairs formed by a label ℓ and a target state s', so that, when we focus on a single label ℓ we can have $\sum_{s' \in S} d(\ell, s') < 1$, while in a simple Segala model, if $(\ell, d) \in \alpha_{\mathsf{sim}}(s)$ then $\sum_{s' \in S} d(s') = 1$.

15.2 We have that s_0 and s_2 are bisimilar, while s_0 and s_1 are not (and therefore also s_2 is not bisimilar to s_1).

The Larsen-Skou formula $\langle 2€ \rangle_{\frac{1}{3}} (\langle coffee \rangle true \wedge \langle beer \rangle true)$ is satisfied by s_1 and not by s_0.

The equivalence relation $R \stackrel{\text{def}}{=} \{ \{s_0, s_2\}, \{s_0', s_2'\}, \{s_0'', s_2'', s_2'''\} \}$ is a bisimulation relation that relates s_0 and s_2. In fact, letting $I_1 = \{s_0, s_2\}$, $I_2 = \{s_0', s_2'\}$ and $I_3 = \{s_0'', s_2'', s_2'''\}$ we have the equalities:

$$\begin{array}{rr}
\gamma(s_0)(2€)(I_2) = \frac{2}{3} & \gamma(s_2)(2€)(I_2) = \frac{2}{3} \\
\gamma(s_0)(2€)(I_3) = \frac{1}{3} & \gamma(s_2)(2€)(I_3) = \frac{1}{3} \\
\gamma(s_0)(3.5€)(I_2) = \frac{1}{3} & \gamma(s_2)(3.5€)(I_2) = \frac{1}{3} \\
\gamma(s_0)(3.5€)(I_3) = \frac{2}{3} & \gamma(s_2)(3.5€)(I_3) = \frac{1}{3} + \frac{1}{3} = \frac{2}{3} \\
\gamma(s_0')(coffee)(I_1) = 1 & \gamma(s_2')(coffee)(I_1) = 1 \\
\gamma(s_0'')(beer)(I_1) = 1 & \gamma(s_2'')(beer)(I_1) = 1 = \gamma(s_2''')(beer)(I_1)
\end{array}$$

where all the omitted cases are assigned null probabilities.

Problems of Chapter 16

16.1 Let $\alpha_{\text{PEPA}} : S \to L \to S \to \mathbb{R}$ be a transition function that assigns the rate $\alpha_{\text{PEPA}}(s)(\ell)(s')$ to any transition $s \stackrel{\ell}{\to} s'$ (it assigns rate 0 when there is no transition from s to s' with label ℓ). We extend the transition function to deal with sets of target states, by defining the function $\gamma_{\text{PEPA}} : S \to L \to \wp(S) \to \mathbb{R}$ as:

$$\gamma_{\text{PEPA}}(s)(\ell)(I) = \sum_{s' \in I} \alpha_{\text{PEPA}}(s)(\ell)(s').$$

Then, we define the function $\Phi_{\text{PEPA}} : \wp(S \times S) \to \wp(S \times S)$ by:

$$\forall s_1, s_2 \in S.\ s_1\ \Phi_{\text{PEPA}}(R)\ s_2 \stackrel{\text{def}}{=} \forall \ell \in L.\ \forall I \in S_{/\equiv_R}.\ \gamma_{\text{PEPA}}(s_1)(\ell)(I) = \gamma_{\text{PEPA}}(s_2)(\ell)(I).$$

Finally, a PEPA bisimulation is a relation R such that $R \subseteq \Phi_{\text{PEPA}}(R)$ and the PEPA bisimilarity $\simeq_{\text{PEPA}}$ is the largest PEPA bisimulation, i.e.:

$$simeq_{\mathrm{PEPA}} \stackrel{\mathrm{def}}{=} \bigcup_{R \subseteq \Phi_{\mathrm{PEPA}}(R)} R.$$

16.2

1. We let:

$$\begin{array}{ll} P \stackrel{\mathrm{def}}{=} (get, rg).P' & R \stackrel{\mathrm{def}}{=} (get, rg').R' \\ P' \stackrel{\mathrm{def}}{=} (task, rt).P & R' \stackrel{\mathrm{def}}{=} (update, ru).R \\ S \stackrel{\mathrm{def}}{=} (P \underset{\varnothing}{\bowtie} P) \underset{\{get\}}{\bowtie} R. & \end{array}$$

2. Since $rg' > 2rg$, when computing the apparent rate of action *get* in S we have:

$$\begin{aligned} r_{get}(S) &= \min\{r_{get}(P \underset{\varnothing}{\bowtie} P), r_{get}(R)\} \\ &= \min\{r_{get}(P) + r_{get}(P), rg'\} \\ &= \min\{2rg, rg'\} = 2rg. \end{aligned}$$

3. The LTS of the system S has eight possible states: