Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

May 16, 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.

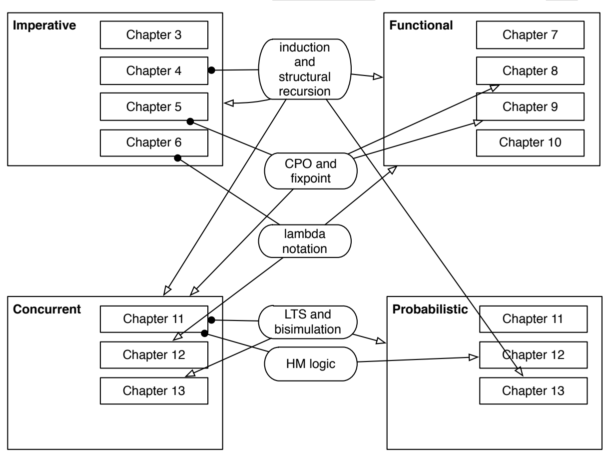
Each of the above models can be studied in separation from the others, but previous parts introduce a body of notions and techniques that are also applied and extended in later parts.

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,
February 2016

Roberto Bruni
Ugo Montanari

Acknowledgements

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

Contents

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\approxeq$	CCS weak observational congruence (see Section 11.8.2)
$\cong$	CCS dynamic bisimilarity (see Definition 11.18)
$\sim_E$	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)
$\mathrm{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.2.2)
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 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 ingenuous, 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

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

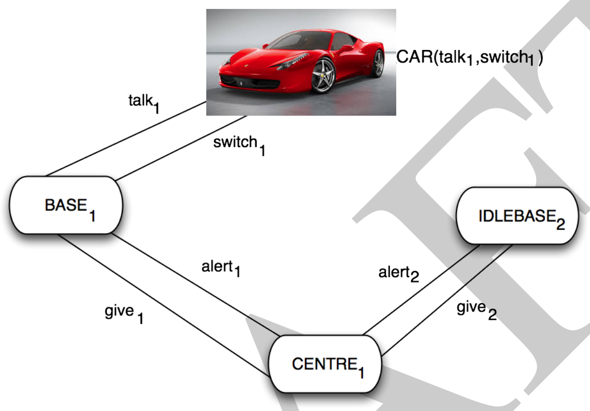
and after the communication it becomes

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

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 $IDLEBASE_i$.

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

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 the name 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.

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

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;
$p\vert q$:	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** $x.\ (x\vert p)$.

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;
$\overline{x}y$:	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 $\overline{x}y.p$ 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 $\overline{x}y.p$ the name x is free in p. As usual, we consider 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).

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

Fig. 13.1: Free names and bound names

Unlike for CCS, the scope of the name y in the restricted process $(y)p$ is not statically determined to coincide with p. In fact, in the π-calculus, channel names are values that can be transmitted, so the process p can send the name y to another process q which thus falls under the scope of y (see Section 13.1). The possibility to enlarge the scope of a restricted name is a very useful feature of the π-calculus, called *name extrusion*. It allows to modify the structure of private communications between agents. Moreover, name extrusion 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 $\mathrm{fn}(\cdot)$, bound names $\mathrm{bn}(\cdot)$ are extended to labels by letting:

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

Moreover, we let $\mathrm{n}(\alpha) \stackrel{\text{def}}{=} \mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha)$ denote the set of names appearing in α.

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.

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

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

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

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

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

13.3.3 Name Matching

Name matching can be used to write a process that receives a name y 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 $login(xp).[xp = pwd].p$.

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

The rule (Match) allows to check the equality of names and to unblock the process p if it is satisfied. If the matching condition is not satisfied we can not continue the execution.

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

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

Instead, we do not allow the transition

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

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

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

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

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

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.

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

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

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

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

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

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 any.

13.3.9 A Sample Derivation

We conclude this section by showing an example of the use of the rule system.

Example 13.3 (Scope extrusion). 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 rule of the operational semantics we can proceed in a goal-oriented fashion to find a derivation for the corresponding transition:

$$
\begin{array}{rl}
\multicolumn{2}{l}{(((y)\bar{x}y.p) \mid q) \mid x(z).r \xrightarrow{\alpha} s} \\
\nwarrow_{(\text{CloseL}),\ \alpha=\tau,\ s=(w)(s_1 \mid r_1)} & ((y)\bar{x}y.p) \mid q \xrightarrow{\bar{x}(w)} s_1, \quad x(z).r \xrightarrow{x(w)} r_1 \\
\nwarrow_{(\text{ParL}),\ s_1=p_1 \mid q,\ w\notin \text{fn}(q)} & (y)\bar{x}y.p \xrightarrow{\bar{x}(w)} p_1 \quad x(z).r \xrightarrow{x(w)} r_1 \\
\nwarrow_{(\text{Open}),\ p_1=p_2[^w/_y],\ w\notin \text{fn}((y).p)} & \bar{x}y.p \xrightarrow{\bar{x}y} p_2, \quad x(z).r \xrightarrow{x(w)} r_1 \\
\overset{*}{\nwarrow}_{(\text{Out})+(\text{In}),\ r_1=r[^w/_z],\ p_2=p,\ w\notin \text{fn}((z).r)} & \square
\end{array}
$$

so we have:

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

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 $w \notin \text{fn}(q) \cup \text{fn}((y)p) \cup \text{fn}((z)r)$.

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

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

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 $x_1, x_2, \ldots, x_k$;
- a finite number of guarded sums[3] $s_1, s_2, \ldots, s_n$;
- and a finite number of processes $p_1, p_2, \ldots, p_m$, 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:

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

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

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

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

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:*

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 that, for any p, p', if $p \xrightarrow{\tau} p'$, then $p \mapsto p'$ by rule induction on $p \xrightarrow{\tau} p'$, from which the thesis follows immediately. $\square$

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:

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

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:

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

[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 deadlock, 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.

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):

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

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 $\mathbf{nil}$. 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):

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

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.

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:

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

The agents p and q are bisimilar (according to both early and late bisimilarities), as they generate isomorphic transition systems. 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

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

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 $p \stackrel{\tau}{\Rightarrow} p' \xrightarrow{\alpha} q' \stackrel{\tau}{\Rightarrow} q$.

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

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

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:

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

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 $\mathrm{fn}(p) \neq \mathrm{fn}(q)$.

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

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.

---

[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 $p = \mathbf{nil}$.

13.8. 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).