Texts in Theoretical Computer Science.
An EATCS Series

Roberto Bruni
Ugo Montanari

Models of Computation

Springer

Texts in Theoretical Computer Science. An EATCS Series

More information about this series at http://www.springer.com/series/3214

Roberto Bruni · Ugo Montanari

Models of Computation

Springer

Roberto Bruni
Dipartimento di Informatica
Università di Pisa
Pisa
Italy

Ugo Montanari
Dipartimento di Informatica
Università di Pisa
Pisa
Italy

ISSN 1862-4499
Texts in Theoretical Computer Science. An EATCS Series
ISBN 978-3-319-42898-7 ISBN 978-3-319-42900-7 (eBook)
DOI 10.1007/978-3-319-42900-7

Library of Congress Control Number: 2017937525

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

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, no. 1, 161–228, 1939.

Foreword

Based on more than fifteen years of teaching, this book provides an exceptionally useful addition to the introductory literature on Models of Computation. It covers a wealth of material on imperative, functional, concurrent and probabilistic computation, their models and reasoning techniques. It does so in a carefully pedagogic way, honed over the years with the help of the students at Pisa. Supplemented with educative examples, explanations and problems (accompanied by solutions) and including a coverage of the techniques of probabilistic and quantitative computation, it provides the student with a rich background and solid foundation for future study and research.

If quality of teaching is to be judged from the quality of students, then teaching in Pisa scores very highly indeed. Over several generations brilliant students from Pisa have made their mark in Computer Science in Europe and the US. All those I know have come under the influence of Ugo Montanari and more recently that of his colleague Roberto Bruni. The broader access this book gives to their expertise and teaching is very welcome.

On a personal note it is a pleasure to acknowledge the friendly collaboration that Ugo Montanari has helped foster over many years, from the first time I visited Pisa for the Pisa Summer School in 1978 — my PhD thesis was born on the grassy area by the Cathedral and Leaning Tower — through to Ugo leading an EU project in which I participated. On the possible start of the UK's withdrawal from the EU I look back with nostalgia to those days of openness and cooperation, now fallen victim to the selfish interference of politicians. This book in its unifying treatment of a criss-cross of work carried out in Europe and the US is a testimony to the power of collaboration in research and education.

Cambridge
July 2016

Glynn Winskel

Preface

The origins of this book have their roots in more than 15 years of teaching a course on formal semantics to Computer Science graduate 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 into *Tecniche di Specifica e Dimostrazione* (*Techniques for Specifications and Proofs*) and finally into 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 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 into 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, and 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, nondeterministic 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 logic 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.

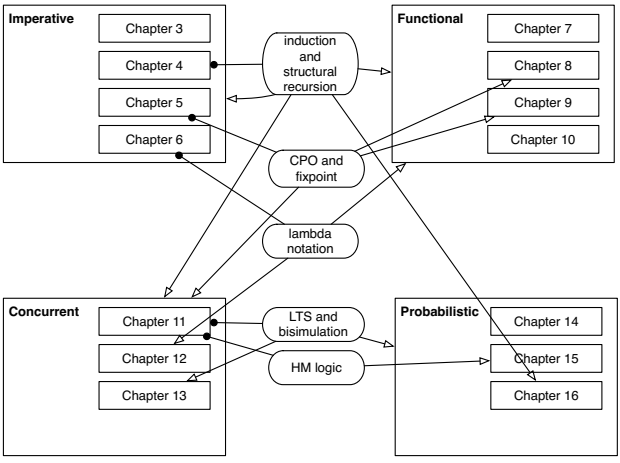
Each of the above models can be studied separately from the others, but previous parts introduce a body of notions and techniques that are also applied and extended in later parts.

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 the π-calculus, are just overviewed. The material in Part III can be used in conjunction with other textbooks, e.g., on model checking or the 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 provides 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 rectangular box for each chapter/part and a round-cornered 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 notions 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 above mentioned bisimilarity is the coarsest bisimulation relation. Various forms of bisimulation are studied in Parts IV and V.
HM-logic:	Hennessy-Milner logic is the logic counterpart of bisimilarity: two states are bisimilar if and only if they satisfy the same set of HM-logic formulas. In the context of probabilistic systems, the approach is extended to Larsen-Skou logic in Chapter 15.

Each chapter of the book concludes with 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 Texts series. We thank the anonymous reviewers of the book for helping us to improve the presentation and remove as many errors and typos as possible: those left are our full responsibility. We thank Ronan Nugent and all the people at Springer for their patience and support: they have done great 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, and 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

Part II IMP: a Simple Imperative Language

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 early bisimilarity (see Definition 13.4)
$\sim_L$	π-calculus 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 Section 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 I
Preliminaries

This part introduces the basic terminology, notation and mathematical tools used in the rest of the book.

Chapter 1
Introduction

It is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct. (Dana Scott)

Abstract This chapter overviews the motivation, guiding principles and main concepts used in the book. It starts by explaining the role of formal semantics and different approaches to its definition, then briefly touches on some important subjects covered in the book, such as domain theory and induction principles, and it is concluded by an explanation of the content of each chapter, together with a list of references to the literature for studying some topics in more depth or for using some companion textbooks in conjunction with the current text.

1.1 Structure and Meaning

Any programming language is fully defined in terms of three essential features:

Syntax:	refers to the appearance of the programs of the language;
Types:	restrict the syntax to enforce suitable properties on programs;
Semantics:	refers to the meanings of (well-typed) programs.

Example 1.1. The alphabet of Roman numerals, the numeric system used in ancient Rome, consists of seven letters drawn from the Latin alphabet. A value is assigned to each letter (see Table 1.1) and a number n is expressed by juxtaposing some letters whose values sum to n. Not all sequences are valid though. Symbols are usually placed from left to right, starting with the largest value and ending with the smallest value. However, to avoid four repetitions in a row of the same letter, e.g., IIII, subtractive notation is used: when a symbol with smaller value u is placed to the left of a symbol with higher value v they represent the number $v - u$. So the symbol I can be placed before V or X; the symbol X before L or C and the symbol C before D or M, and 4 is written IV instead of IIII. While IX and XI are both correct sequences, with values 9 and 11, respectively, the sequence IXI is not correct and has no corresponding value. The rules that prescribe the correct sequences of symbols define the (well-typed) syntax of Roman numerals. The rules that define how to evaluate Roman numerals to positive natural numbers give their semantics.

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_1

Table 1.1: Alphabet of Roman numerals

Symbol	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1,000

1.1.1 Syntax, Types and Pragmatics

The syntax of a *formal* language tells us which sequences of symbols are valid statements and which ones make no sense and should be discarded.

Mathematical tools such as regular expressions, context-free grammars and Backus-Naur Form (BNF) are now widely applied tools for defining the syntax of formal languages. They are studied in every computer science degree and are exploited in technical appendices of many programming language manuals to define the grammatical structure of programs without ambiguities.

Types can be used to limit the occurrence of errors or to allow compiler optimisations or to reduce the risk of introducing bugs or just to discourage certain programming malpractices. Types are often presented as sets of logic rules, called *type systems*, that are used to assign a type unambiguously to each program and computed value. Different type systems can be defined over the same language to enforce different properties.

However, grammars and types do not explain what a correctly written program means. Thus, every language manual also contains natural language descriptions of the meaning of the various constructs, how they should be used and styled, and example code fragments. This attitude falls under the *pragmatics* of a language, describing, e.g., how the various features should be used and which auxiliary tools are available (syntax checkers, debuggers, etc.). Unfortunately this leaves space for different interpretations that can ultimately lead to discordant implementations of the same language or to compilers that rely on questionable code optimisation strategies.

If an official formal semantics of a language were available as well, it could accompany the language manual too and solve any ambiguity for implementors and programmers. This is not yet the case because effective techniques for specifying the run-time behaviour of programs in a rigorous manner have proved much harder to develop than grammars.

1.1.2 Semantics

The origin of the word ‘semantics’ can be traced back to a book by French philologist Michel Bréal (1832–1915), published in 1900, where it referred to *the study of how words change their meanings*. Subsequently, the word ‘semantics’ has also changed its meaning, and it is now generally defined as *the study of the meanings of words and phrases in a language*.

In Computer Science, semantics is concerned with *the study of the meaning of (well-typed) programs*.

Studies in formal semantics are not always easily accessible to a student of computer science or engineering without a good background in mathematical logic, and, as a consequence, they are often regarded as an esoteric subject by people not familiar enough with the mathematical tools involved.

Following [36] we can ask ourselves: *what do we gain by formalising the semantics of a programming language?*

After all, programmers can write programs that are trusted to "work as expected" once they have been thoroughly tested, so how would the effort spent in a rigorous formalisation of the semantics pay back? An easy answer is that today, in the era of the Internet of Things and cyberphysical systems, our lives, the machines and devices we use, and the entire world run on software. It is not enough to require that medical implants, personal devices, planes and nuclear reactors seem to "work as expected"!

To give a more circumstantiated answer, we can start from the related question: *what was gained when language syntax was formalised?*

It is generally understood that the formalisation of syntax leads, e.g., to the following benefits:

1. it standardises the language; this is crucial
   - to programmers, as a guide to write syntactically correct programs, and
   - to implementors, as a reference to develop a correct parser.
2. it permits a formal analysis of many properties, such as finding and resolving parsing ambiguities.
3. it can be used as input to a compiler front-end generating tool, such as Yacc, Bison, or Xtext. In this way, from the syntax definition one can automatically derive an implementation of the compiler's front end.

Formalising the semantics of a programming language can then lead to similar benefits:

1. it standardises a machine-independent specification of the language; this is crucial:
   - to programmers, for improving the programs they write, and
   - to implementors, to design a correct and efficient code generator.
2. it permits a formal analysis of program properties, such as type safety, termination, specification compliance and program equivalence.
3. it can be used as input to a compiler back-end generating tool. In this way, the semantics definition also gives the (prototypal and possibly inefficient) implementation of the back end of the language's compiler. Moreover, efficient compilers need to adhere to the semantics and their optimisations need correctness proofs.

What is then the semantics of a programming language?

A crude view is that the semantics of a programming language is defined by (the back end of) its compiler or interpreter: from the source program to the target code executed by the computer. This view is clearly not acceptable because, e.g., it refers

to specific pieces of commercial hardware and software, and the specification is not good for portability, it is not at the right level of abstraction to be understood by a programmer, it is not at the right level of abstraction to state and prove interesting properties of programs (for example, two programs written for the same purpose by different programmers are likely different, even if they should have the same meaning). Finally, if different implementations are given, how do we know that they are correct and compatible?

Example 1.2. We can hardly claim to know that two programs mean the same thing if we cannot tell what a program means. For example, consider the Java expressions

```
x + (y + z)                    (x + y) + z
```

Are they equivalent? Can we replace the former with the latter (and vice versa) in a program, without changing its meaning? Under what circumstances?[1]

To give a semantics for a programming language means to define the behaviour of any program written in this language. As there are infinitely many programs, one would like to have a finitary description of the semantics that can take into account any of them. Only when the semantics is given can one prove such important properties as program equivalence or program correctness.

1.1.3 Mathematical Models of Computation

In giving a formal semantics to a programming language we are concerned with *building a mathematical model*: its purpose is to serve as a basis for understanding and reasoning about how programs behave. Not only is a mathematical model useful for various kinds of analysis and verification, but also, at a more fundamental level, because the activity of trying to define precisely the meanings of program constructions can reveal all kinds of subtleties which it is important to be aware of.

Unlike the acceptance of BNF as a standard definition method for syntax, there is little hope that a single definition method will take hold for semantics. This is because semantics

- is harder to formalise than syntax,
- has a wider variety of applications,
- is dependent on the properties we want to tackle, i.e., different models are best suited for tackling different issues.

In fact, different semantic styles and models have been developed for different purposes. The overall aim of the book is to study the main semantic styles, compare their expressiveness, and apply them to study program properties. To this aim it is

[1] Recall that '+' is overloaded in Java: it sums integers and floating points and it concatenates strings.

fundamental to gain acquaintance with the principles and theories on which such semantic models are based.

Classically, semantics definition methods fall roughly into three groups: operational, denotational and axiomatic. In this book we will focus mainly on the first two kinds of semantics, which find wider applicability.

Operational Semantics

In the operational semantics it is of interest *how* the effect of a computation is achieved. Some kind of abstract machine[2] is first defined, then the operational semantics describes the meaning of a program in terms of the steps/actions that this machine executes. The focus of operational semantics is thus on states and state transformations.

An early notable example of operational semantics was concerned with the semantics of LISP (LISt Processor) by John McCarthy (1927–2011) [19]. A later example was the definition of the semantics of Algol 68 over a hypothetical computer [42].

In 1981, Gordon Plotkin (1946–) introduced the structural operational semantics style (SOS-style) in the technical report [28] which is still one of the most-cited technical reports in computer science, only recently revised and re-issued in a journal [30, 31].

Gilles Kahn (1946–2006) introduced another form of operational semantics, called natural semantics, or big-step semantics, in 1987, where possibly many steps of execution are incorporated into a single logical derivation [16].

It is relatively easy to write the operational semantics in the form of Horn clauses, a particular form of logical implications. In this way, they can be interpreted by a logic programming system, such as Prolog.[3]

Because of the strong connection with the syntactic structure and the fact that the mathematics involved is usually less complicated than in other semantic approaches, SOS-style operational semantics can provide programmers with a concise and accurate description of what the language constructs do. In fact, it is syntax oriented, inductive and easy to grasp. Operational semantics is also versatile: it applies with minor variations to most different computing paradigms.

---

[2] The term *machine* ordinarily refers to a *physical* device that performs mechanical functions. The term *abstract* distinguishes a physically existent device from one that exists in the imagination of its inventor or user: it is a convenient conceptual abstraction that leaves out many implementation details. The archetypical abstract machine is the Turing machine.

[3] However, we have to leave aside issues about performance and the fact that Prolog is not complete, because it exploits a depth-first exploration strategy for the next step to execute: backtracking out of wrong attempted steps is only possible if they are finitely many.

Denotational Semantics

In denotational semantics, the meaning of a well-formed program is some mathematical object (e.g., a function from input data to output data). The steps taken to calculate the output and the abstract machine where they are executed are unimportant: only the *effect* is of interest, not how it is obtained.

The essence of denotational semantics lies in the principle of *compositionality*: the semantics takes the form of a function that assigns an element of some mathematical domain to each individual construct, in such a way that *the meaning of a composite construct does not depend on the particular form of the constituent constructs, but only on their meanings.*

Denotational semantics originated in the pioneering work of Christopher Strachey (1916–1975) and Dana Scott (1932–) in the late 1960s and in fact it is sometimes called Scott-Strachey semantics [37, 39, 38].

Denotational semantics descriptions can also be used to derive implementations. Still there is a problem with performance: operations that can be efficiently performed on computer hardware, such as reading or changing the contents of storage cells, are first mapped to relatively complicated mathematical notions which must then be mapped back again to a concrete computer architecture.

One limitation is that in the case of concurrent, interactive, nondeterministic systems the body of mathematics involved in the definition of denotational semantics is quite heavy.

Axiomatic Semantics

Instead of directly assigning a meaning to each program, axiomatic semantics gives a description of the constructs in a programming language by providing logical conditions that are satisfied by these constructs. Axiomatic semantics places the focus on valid *assertions* for proving program correctness: there may be aspects of the computation and of the effect that are deliberately ignored.

The axiomatic semantics has been put forward by the work of Robert W. Floyd (1936–2001) on flowchart languages [8] and of Tony Hoare (1934–) on structured imperative programs [15]. In fact it is sometimes referred to as Floyd-Hoare logic. The basic idea is that program statements are described by two logical assertions: a pre-condition, prescribing the state of the system before executing the program, and a post-condition, satisfied by the state after the execution, when the pre-condition is valid. Using such an axiomatic description it is possible, at least in principle, to prove the correctness of a program with respect to a specification. Two main forms of correctness are considered:

Partial: a program is partially correct w.r.t. a pre-condition and a post-condition if whenever the initial state fulfils the pre-condition *and* the program terminates, the final state is guaranteed to fulfil the post-condition. The partial correctness property does not ensure that the program will terminate; e.g., a program which never terminates satisfies every property.

Total: a program is totally correct w.r.t. a pre-condition and a post-condition if whenever the initial state fulfils the pre-condition, *then* the program terminates, and the final state is guaranteed to fulfil the post-condition.

The axiomatic method becomes cumbersome in the presence of modular program constructs, e.g., goto's and objects, but also as simple as blocks and procedures. Another limitation of axiomatic semantics is that it is scarcely applicable to the case of concurrent, interactive systems, whose correct behaviour often involves non-terminating computations (for which post-conditions cannot be used).

1.2 A Taste of Semantic Methods: Numerical Expressions

We can give a first, informal overview of the different flavours of semantic styles we will consider in this book by taking a simple example of numerical expressions.[4] Let us consider two syntactic categories *Nums* and *Exp*, respectively, for numerals $\mathtt{n} \in \mathit{Nums}$ and expressions $E \in \mathit{Exp}$, defined by the grammar

$$\begin{array}{lcl} \mathtt{n} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\ e & ::= & \mathtt{n} \mid e \oplus e \mid e \otimes e \end{array}$$

The above language of numerical expressions uses the auxiliary set of *numerals*, *Nums*, which are syntactic representations of the more abstract set of natural numbers.

Remark 1.1 (Numbers vs numerals). The natural numbers $0, 1, 2, \ldots$ are mathematical objects which exist in some abstract world of concepts. They find concrete representations in different languages. For example, the number 5 is represented by

- the string "five" in English,
- the string "101" in binary notation,
- the string "V" in Roman numerals.

To differentiate between numerals (5) and numbers (5) we use here different fonts.

From the grammar it is evident that there are three ways to build expressions:

- any numeral n is also an expression;
- if we are given any two expressions e_0 and e_1, then $e_0 \oplus e_1$ is also an expression;
- if we are given any two expressions e_0 and e_1, then $e_0 \otimes e_1$ is also an expression.

In the book we will always use abstract syntax representations, as if all concrete terms were parsed before we start to work with them.

Remark 1.2 (Concrete and abstract syntax). While the *concrete* syntax of a language is concerned with the precise linear sequences of symbols which are valid terms of

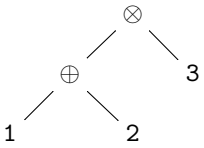
[4] The example has been inspired by some course notes on the "Semantics of programming languages", by Matthew Hennessy.

the language, we are interested in the *abstract* syntax, which describes expressions purely in terms of their structure. We will never be worried about where the brackets are in expressions like

$$1 \oplus 2 \otimes 3$$

because we will never deal with such unparsed terms.

In other words we are considering only (valid) *abstract syntax trees*, like



Since it would be tedious to draw trees every time, we use linear syntax and brackets, like $(1 \oplus 2) \otimes 3$, to save space while avoiding ambiguities.

An Informal Semantics

Since in the expressions we deliberately used some non-standard symbols $\oplus$ and $\otimes$, we must define what is their meaning. Programmers primarily learn the semantics of a language through examples, their intuitions about the underlying computational model, and some natural language description. An informal description of the meaning of the expressions we are considering could be the following:

- a numeral $\mathtt{n}$ is evaluated to the corresponding natural number n;
- to find the value associated with an expression of the form $e_0 \oplus e_1$ we evaluate the expressions e_0 and e_1 and take the sum of the results;
- to find the value associated with an expression of the form $e_0 \otimes e_1$ we evaluate the expressions e_0 and e_1 and take the product of the results.

We hope the reader agrees that the above guidelines are sufficient to determine the value of any well-formed expression, no matter how large.[5]

To accompany the description with some examples, we can add that

- 2 is evaluated to 2;
- $(1 \oplus 2) \otimes 3$ is evaluated to 9;
- $(1 \oplus 2) \otimes (3 \oplus 4)$ is evaluated to 21.

Since natural language is notoriously prone to mis-interpretations and mis-understandings, in the following we try to make the above description more accurate.

We show next how the operational semantics can formalise the steps needed to evaluate an expression over some abstract computational device and how the denotational semantics can assign meaning to numerical expressions (their valuation).

[5] Note that we are not telling the order in which e_0 and e_1 must be evaluated: is it important?

$$\frac{}{\mathtt{n}_0 \oplus \mathtt{n}_1 \to \mathtt{n}}\ n = n_0 + n_1\ (sum) \qquad \frac{e_0 \to e_0'}{e_0 \oplus e_1 \to e_0' \oplus e_1}\ (sumL) \qquad \frac{e_1 \to e_1'}{e_0 \oplus e_1 \to e_0 \oplus e_1'}\ (sumR)$$

$$\frac{}{\mathtt{n}_1 \otimes \mathtt{n}_2 \to \mathtt{n}}\ n = n_1 \times n_2\ (prod) \qquad \frac{e_0 \to e_0'}{e_0 \otimes e_1 \to e_0' \otimes e_1}\ (prodL) \qquad \frac{e_1 \to e_1'}{e_0 \otimes e_1 \to e_0 \otimes e_1'}\ (prodR)$$

Fig. 1.1: Small-step semantics rules for numerical expressions

A Small-Step Operational Semantics

There are several versions of operational semantics for the above language of expressions. The first one we present is likely familiar to you: it simplifies expressions until a value is met. This is achieved by defining judgements of the form

$$e_0 \to e_1$$

to be read as: *after performing one step of evaluation of e_0, the expression e_1 remains to be evaluated.*

Small-step semantics formally describes how individual steps of a computation take place on an abstract device, but it ignores details such as the use of registers and storage addresses. This makes the description independent of machine architectures and implementation strategies.

The logic inference rules are written in the general form (see Section 2.2)

$$\frac{\textit{premises}}{\textit{conclusion}}\ \textit{side-condition} \quad (\textit{rule name})$$

meaning that if the *premises* and the *side-condition* are met then the *conclusion* can be drawn, where the premises consist of one, none or more judgements and the side-condition is a single boolean predicate. The *rule name* is just a convenient label that can be used to refer to the rule. Rules with no premises are called axioms and their conclusion is postulated to be always valid.

The rules for the expressions are given in Figure 1.1. For example, the rule *sum* says that $\oplus$ applied to two numerals evaluates to the numeral representing the sum of the two arguments, while the rule *sumL* (respectively, *sumR*) says that we are allowed to simplify the left (resp., right) argument. Analogously for product.

For example, we can derive both the judgements

$$(1 \oplus 2) \otimes (3 \oplus 4) \to 3 \otimes (3 \oplus 4) \qquad (1 \oplus 2) \otimes (3 \oplus 4) \to (1 \oplus 2) \otimes 7$$

as witnessed by the formal derivations

$$\frac{\dfrac{}{1 \oplus 2 \to 3}\ 3 = 1 + 2\ (sum)}{(1 \oplus 2) \otimes (3 \oplus 4) \to 3 \otimes (3 \oplus 4)}\ (prodL) \qquad \frac{\dfrac{}{3 \oplus 4 \to 7}\ 7 = 3 + 4\ (sum)}{(1 \oplus 2) \otimes (3 \oplus 4) \to (1 \oplus 2) \otimes 7}\ (prodR)$$

A derivation is represented as an (inverted) tree, with the goal to be verified at the root. The tree is generated by applications of the defining rules, with the terminating leaves being generated by axioms. As derivations tend to grow large, we will introduce a convenient alternative notation for them in Chapter 2 (see Example 2.5 and Section 2.3) and will use it extensively in the subsequent chapters.

Note that even for a deterministic program, there can be many different computation sequences leading to the same final result, since the semantics may not specify a totally ordered sequence of evaluation steps.

If we want to enforce a specific evaluation strategy, then we can change the rules to guarantee, e.g., that the leftmost occurrence of an operator $\oplus/\otimes$ which has both its operands already evaluated is always executed first, while the evaluation of the second operand is conducted only after the first operand has been evaluated. We show only the two rules that need to be changed (changes are highlighted with boxes):

$$\frac{e_1 \to e_1'}{\boxed{\mathtt{n}_0} \oplus e_1 \to \boxed{\mathtt{n}_0} \oplus e_1'} \ (sumR) \qquad \frac{e_1 \to e_1'}{\boxed{\mathtt{n}_0} \otimes e_1 \to \boxed{\mathtt{n}_0} \otimes e_1'} \ (prodR)$$

Now the step judgement

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}$$

is no longer derivable.

Instead, it is not difficult to derive the judgements

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \qquad \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes \mathtt{7} \qquad \mathtt{3} \otimes \mathtt{7} \to \mathtt{21}$$

The steps can be composed: let us write

$$e_0 \to^k e_k$$

if e_0 can be reduced to e_k in k steps: that is there exist $e_1, e_2, ..., e_{k-1}$ such that we can derive the judgements

$$e_0 \to e_1 \qquad e_1 \to e_2 \qquad ... \qquad e_{k-1} \to e_k$$

This includes the case when $k = 0$: then e_k must be the same as e_0, i.e., in zero steps any expression can reduce to itself.

In our example, by composing the above steps, we have

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to^3 \mathtt{21}$$

We also write

$$e \not\to$$

when no expression e' can be found such that $e \to e'$.

It is immediate to see that for any numeral $\mathtt{n}$, we have $\mathtt{n} \not\to$, as no conclusion of the inference rules has a numeral as the source of the transition.

To fully evaluate an expression, we need to indefinitely compute successive derivations until eventually a final numeral is obtained that cannot be evaluated further. We write

$$e \to^* \mathtt{n}$$

to mean that there is some natural number k such that $e \to^k \mathtt{n}$, i.e., e can be evaluated to $\mathtt{n}$ in k steps. The relation $\to^*$ is called the *reflexive and transitive closure of* $\to$. Note that we have, e.g., $\mathtt{n} \to^* \mathtt{n}$ for any numeral $\mathtt{n}$.

In our example we can derive the judgement

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to^* \mathtt{21}$$

Small-step operational semantics will be especially useful in Parts IV and V to assign different semantics to non-terminating systems.

A Big-Step Operational Semantics (or Natural Semantics)

Like small-step semantics, a natural semantics is a set of inference rules, but a *complete computation is done as a single, large derivation*. For this reason, a natural semantics is sometimes called a *big-step operational semantics*.

Big-step semantics formally describes how the overall results of the executions are obtained. It hides even more details than the small-step operational semantics. Like small-step operational semantics, natural semantics shows the context in which a computation step occurs, and like denotational semantics, natural semantics emphasises that the computation of a phrase is built from the computations of its sub-phrases.

Natural semantics have the advantage of often being simpler (needing fewer inference rules) and of often directly corresponding to an efficient implementation of an interpreter for the language. In our running example, we disregard the individual steps that lead to the result and focus on the final outcome, i.e., we formalise the predicate $e \to^* \mathtt{n}$. Typically, the same predicate symbol $\to$ is used also in the case of natural semantics. To avoid ambiguities and to not overload the notation, here, for the sake of the running example, we use a different symbol. We define the predicate

$$e \twoheadrightarrow \mathtt{n}$$

to be read as *the expression e is (eventually) evaluated to* $\mathtt{n}$.

The rules are reported in Figure 1.2. This time only three rules are needed, which immediately correspond to the informal semantics we gave for numerical expressions.

We can now verify that the judgement

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \twoheadrightarrow \mathtt{21}$$

can be derived as follows:

$$\frac{}{\mathtt{n} \twoheadrightarrow \mathtt{n}}\ (num) \qquad \frac{e_0 \twoheadrightarrow \mathtt{n}_1 \quad e_1 \twoheadrightarrow \mathtt{n}_2}{e_0 \oplus e_1 \twoheadrightarrow \mathtt{n}}\ n = n_1 + n_2\ (sum) \qquad \frac{e_0 \twoheadrightarrow \mathtt{n}_1 \quad e_1 \twoheadrightarrow \mathtt{n}_2}{e_0 \otimes e_1 \twoheadrightarrow \mathtt{n}}\ n = n_1 \times n_2\ (prod)$$

Fig. 1.2: Natural semantics for numerical expressions

$$\dfrac{\dfrac{\dfrac{}{1 \twoheadrightarrow 1}\ (num) \quad \dfrac{}{2 \twoheadrightarrow 2}\ (num)}{1 \oplus 2 \twoheadrightarrow 3}\ 3 = 1+2\ (sum) \quad \dfrac{\dfrac{}{3 \twoheadrightarrow 3}\ (num) \quad \dfrac{}{4 \twoheadrightarrow 4}\ (num)}{3 \oplus 4 \twoheadrightarrow 7}\ 7 = 3+4\ (sum)}{(1 \oplus 2) \otimes (3 \oplus 4) \twoheadrightarrow 21}\ 21 = 3 \times 7\ (prod)$$

Small-step operational semantics gives more control of the details and order of evaluation. These properties make small-step semantics more convenient when proving type soundness of a type system against an operational semantics. Natural semantics can lead to simpler proofs, e.g., when proving the preservation of correctness under some program transformation. Natural semantics is also very useful to define reduction to canonical forms.

An interesting drawback of natural semantics is that semantics derivations can be drawn only for terminating programs. The main disadvantage of natural semantics is thus that non-terminating (diverging) computations do not have an inference tree.

We will exploit natural semantics mainly in Parts II and III of the book.

A Denotational Semantics

Differently from operational semantics, denotational semantics is concerned with manipulating mathematical objects and not with executing programs.

In the case of expressions, the intuition is that a term represents a number (expressed in the form of a calculation). So we can choose as a *semantic domain* the set of natural numbers $\mathbb{N}$, and the *interpretation function* will then map expressions to natural numbers.

To avoid ambiguities between pieces of syntax and mathematical objects, we usually enclose syntactic terms within a special kind of brackets $[\![\cdot]\!]$ that serve as a separation. It is also common, when different interpretation functions are considered, to use calligraphic letters to distinguish the kind of terms they apply to (one for each syntactical category).

In our running example, we define two semantic functions:

$$\mathscr{N}[\![\cdot]\!] : \mathit{Nums} \to \mathbb{N}$$
$$\mathscr{E}[\![\cdot]\!] : \mathit{Exp} \to \mathbb{N}$$

Remark 1.3. When we study more complex languages, we will need to involve more complex (and less familiar) domains than $\mathbb{N}$. For example, as originally developed by Strachey and Scott, denotational semantics provides the meaning of a computer program as a function that maps input into output. To give denotations to recur-

sively defined programs, Scott proposed working with continuous functions between domains, specifically complete partial orders.

Notice that our choice of semantic domain has certain immediate consequences for the semantics of our language: it implies that every expression will mean exactly one number! Without having defined yet the interpretation functions, and contrary to the operational semantics definitions, anyone looking at the semantics already knows that the language is

deterministic: each expression has at most one answer;
normalising: every expression has an answer.

Giving a meaning to numerals is immediate

$$\mathscr{N}[\![\mathtt{n}]\!] = n$$

For composite expressions, the meaning will be determined by composing the meaning of the arguments

$$\begin{aligned}\mathscr{E}[\![\mathtt{n}]\!] &= \mathscr{N}[\![\mathtt{n}]\!]\\ \mathscr{E}[\![e_0 \oplus e_1]\!] &= \mathscr{E}[\![e_0]\!] + \mathscr{E}[\![e_1]\!]\\ \mathscr{E}[\![e_0 \otimes e_1]\!] &= \mathscr{E}[\![e_0]\!] \times \mathscr{E}[\![e_1]\!]\end{aligned}$$

We have thus defined the interpretation function *by induction on the structure of the expressions* and it is

compositional: the meaning of complex expressions is defined in terms of the meaning of their constituents.

As an example, we can interpret our running expression:

$$\begin{aligned}\mathscr{E}[\![(1 \oplus 2) \otimes (3 \oplus 4)]\!] &= \mathscr{E}[\![1 \oplus 2]\!] \times \mathscr{E}[\![3 \oplus 4]\!]\\ &= (\mathscr{E}[\![1]\!] + \mathscr{E}[\![2]\!]) \times (\mathscr{E}[\![3]\!] + \mathscr{E}[\![4]\!])\\ &= (\mathscr{N}[\![1]\!] + \mathscr{N}[\![2]\!]) \times (\mathscr{N}[\![3]\!] + \mathscr{N}[\![4]\!])\\ &= (1+2) \times (3+4) = 21\end{aligned}$$

Denotational semantics is best suited for sequential systems and thus exploited in Parts II and III.

Semantic Equivalence

We have now available three different semantics for numerical expressions:

$$e \to^* \mathtt{n} \qquad e \twoheadrightarrow \mathtt{n} \qquad \mathscr{E}[\![e]\!]$$

and we are faced with several questions:

1. Is it true that for every expression e there exists some numeral $\mathtt{n}$ such that $e \to^* \mathtt{n}$?
   The same property, often referred to as *normalisation*, can be asked also for $e \twoheadrightarrow \mathtt{n}$, while it is trivially satisfied by $\mathscr{E}[\![e]\!]$.
2. Is it true that if $e \to^* \mathtt{n}$ and $e \to^* \mathtt{m}$ we have $\mathtt{n} = \mathtt{m}$?
   The same property, often referred to as *determinacy*, can be asked also for $e \twoheadrightarrow \mathtt{n}$, while it is trivially satisfied by $\mathscr{E}[\![e]\!]$.
3. Is it true that $e \to^* \mathtt{n}$ iff $e \twoheadrightarrow \mathtt{n}$?
   This has to do with the *consistency* of the semantics and the question can be posed between any two of the three semantics we have defined.

We can also derive some intuitive relations of *equivalence* between expressions:

- Two expressions e_0 and e_1 are equivalent if for any numeral $\mathtt{n}$, $e_0 \to^* \mathtt{n}$ iff $e_1 \to^* \mathtt{n}$.
- Two expressions e_0 and e_1 are equivalent if for any numeral $\mathtt{n}$, $e_0 \twoheadrightarrow \mathtt{n}$ iff $e_1 \twoheadrightarrow \mathtt{n}$.
- Two expressions e_0 and e_1 are equivalent if $\mathscr{E}[\![e_0]\!] = \mathscr{E}[\![e_1]\!]$.

Of course, if we prove the consistency of the three semantics, then we can conclude that the three notions of equivalence coincide.

Expressions with Variables

Suppose now we want to extend numerical expressions with the possibility to include formal parameters in them, drawn from an infinite set X, ranged over by x.

$$e \quad ::= \quad x \quad | \quad \mathtt{n} \quad | \quad e \oplus e \quad | \quad e \otimes e$$

How can we evaluate an expression like $(x \oplus 4) \otimes y$? We cannot, unless the values assigned to x and y are known: in general, the result will depend on them.

Operationally, we must provide such information to the machine, e.g., in the form of some memory $\sigma : X \to \mathbb{N}$ that is part of the machine state. We use the notation $\langle e, \sigma \rangle$ to denote the state where e is to be evaluated in the memory σ. The corresponding small-/big-step rules for variables would then look like

$$\frac{}{\langle x, \sigma \rangle \to \mathtt{n}} \; n = \sigma(x) \; (var) \qquad \frac{}{\langle x, \sigma \rangle \twoheadrightarrow \mathtt{n}} \; n = \sigma(x) \; (var)$$

Exercise 1.1. The reader may complete the missing rules as an exercise.

Denotationally, the interpretation function needs to receive a memory as an additional argument:

$$\mathscr{E}[\![\cdot]\!] : Exp \to ((X \to \mathbb{N}) \to \mathbb{N})$$

Note that this is quite different from the operational approach, where the memory is part of the state.

The corresponding defining equations would then look like

$$\begin{aligned}
\mathscr{E}[\![\mathtt{n}]\!]\sigma &= \mathscr{N}[\![\mathtt{n}]\!] \\
\mathscr{E}[\![x]\!]\sigma &= \sigma(x) \\
\mathscr{E}[\![e_0 \oplus e_1]\!]\sigma &= \mathscr{E}[\![e_0]\!]\sigma + \mathscr{E}[\![e_1]\!]\sigma \\
\mathscr{E}[\![e_0 \otimes e_1]\!]\sigma &= \mathscr{E}[\![e_0]\!]\sigma \times \mathscr{E}[\![e_1]\!]\sigma
\end{aligned}$$

Semantic equivalences must then take into account all the possible memories where expressions are evaluated. To say that e_0 is denotationally equivalent to e_1 we must require that *for any memory* $\sigma : X \to \mathbb{N}$ we have $\mathscr{E}[\![e_0]\!]\sigma = \mathscr{E}[\![e_1]\!]\sigma$.

Exercise 1.2. The reader is invited to restate the consistency between the various semantics and the operational notions of equivalences between expressions taking memories into account.

1.3 Applications of Semantics

Whatever care is taken to make a natural language description of programming languages precise and unambiguous, there always remain some points that are open to several different interpretations. Formal semantics can provide a useful basis for the language design, its implementation, and the analysis and verification of programs.

In the following we summarise some benefits for each of the above categories.

Language Design

The worst form of design errors are cases where the language behaves in a way that is not expected and even less desired by its designers. Fixing a formal semantics for a language is the best way of detecting weak points in the language design itself. Starting from the natural language descriptions of the various features, subtle ambiguities, inconsistencies, complexities and anomalies will emerge, and better ways of dealing with each feature will be discovered.

While the presence of problems can be demonstrated by exhibiting example programs, their absence can only be proved by exploiting a formal semantics. Operational semantics, denotational semantics and axiomatic semantics, in this order, are increasingly sensitive tools for detecting problems in language design.

Increasingly, language designers are using semantics definitions to formalise their creations. Famous examples include Ada [6], Scheme [17] and ML [22]. A more recent witness is the use of Horn clauses to specify the type checker in the Java Virtual Machine version 7.[6]

[6] `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`

Implementation

Semantics can be used to validate prototype implementations of programming languages, to verify the correctness of code analysis techniques exploited in the implementation, like type checking, and to certify many useful properties, such as the correctness of compiler optimisations.

A common phenomenon is the presence of underspecified behaviour in certain circumstances. In practice, such underspecified behaviours can reduce programs' portability from one platform to another.

Perhaps the most significant application of operational semantics definitions is the straightforward generation of prototypal implementations, where the behaviour of programs can be simulated and tested, even if the underlying interpreter can be inefficient. Denotational semantics can also provide a good starting point for automatic language implementation. Automatic generation of implementations is not the only way in which formal semantics can help implementors. If a formal model is available, then hand-crafted implementations can be related to the formal semantics, e.g., to guarantee their correctness.

Analysis and Verification

Semantics offers the main support for reasoning about programs, specifications, implementations and their properties, both mechanically and by hand. It is the unique means to state that an implementation conforms to a specification, or that two programs are equivalent or that a model satisfies some property.

For example, let us consider the following OCaml-like functions:

```
let rec fib n = match n with
                      0 -> 0
                    | 1 -> 1
                    | x -> fib (x-1) + fib (x-2)


let fib n = let rec faux a b cnt = match cnt with
                                        0 -> b
                                      | x -> faux (a+b) a (x-1)
            in faux 1 0 n
```

The second program offers a much more efficient version of the Fibonacci numbers calculation (the number of recursive calls is linear in n, as opposed to the first program where the number of recursive calls is exponential in n). If the two versions can be proved equivalent from the functional point of view, then we can safely replace the first version with the better-performing one.

Synergy Between Different Semantics Approaches

It would be wrong to view different semantics styles as in opposition to each other. They each have their uses and their combined use is more than the sum of its parts. Roughly

- A clear operational semantics is very helpful in implementation and in proving program and language properties.
- Denotational semantics provides the deepest insights to the language designer, being sustained by a rich mathematical theory.
- Axiomatic semantics can lead to strikingly elegant proof systems, useful in developing as well as verifying programs.[7]

As discussed above, the objective of the book is to present different models of computation, their programming paradigms, their mathematical descriptions and some formal analysis techniques for reasoning about program properties. We shall focus on the operational and denotational semantics.

A long-standing research topic is the relationship between the different forms of semantic definitions. For example, while the denotational approach can be convenient when reasoning about programs, the operational approach can drive the implementation. It is therefore of interest whether a denotational definition is equivalent to an operational one.

In mathematical logic, one uses the concepts of soundness and completeness to relate a logic's proof system to its interpretation, and in semantics there are similar notions of soundness and adequacy to relate one semantics to another.

We show how to relate different kinds of semantics and program equivalences, reconciling whenever possible the operational, denotational and logic views by proving some relevant correspondence theorems. Moreover, we discuss the fundamental ideas and methods behind these approaches.

The operational semantics fixes an abstract and concise operational model for the execution of a program (in a given environment). We define the execution as a proof in some logical system that justifies how the effect is achieved, and once we are at this formal level it will be easier to prove properties of the program.

The denotational semantics describes an explicit *interpretation function* over a mathematical domain. The interpretation function for a typical imperative language is a mapping that, given a program, returns a function from any initial state to the corresponding final state, if any (as programs may not terminate). We cover mostly basic cases, without delving into the variety of options and features that are available to the language designer.

The correspondence is well exemplified over the first two paradigms we focus on: a simple IMPerative language called IMP, and a Higher-Order Functional Language called HOFL. For both of them we define what are the programs and in the case of HOFL we also define what are the infinitely many *types* we can handle. Then, we

[7] Axiomatic semantics is mostly directed towards the programmer, but its wide application is complicated by the fact that it is often difficult (more than denotational semantics) to give a clear axiomatic semantics to languages that were not designed with this in mind.

define their operational semantics, their denotational semantics and finally, to some extent, we prove the correspondence between the two.

As explained later in more detail, in the case of the last two paradigms we consider in the monograph, for concurrent and probabilistic systems, the denotational semantics becomes more complex and we replace its role by suitable logics: two systems are then considered equivalent if they satisfy exactly the same formulas in the logic. Also the perspective of the operational semantics is shifted from the computation of a result to the observable interactions with the environment and two systems are considered equivalent if they exhibit the same behaviour (this equivalence is called *abstract semantics*). Nicely, the behavioural equivalence induced by the operational semantics can be shown to coincide with the logical equivalence above.

1.4 Key Topics and Techniques

1.4.1 Induction and Recursion

Proving existential statements can be done by exhibiting a specific witness, but proving universally quantified statements is more difficult, because all the elements must be considered (for disproof, we can again exhibit a single counterexample) and there can be infinitely many elements to check.

The situation is improved when the elements are generated in some finitary way. For example

- any natural number n can be obtained by taking the successor of 0 for n times;
- any well-formed program is obtained by repeated applications of the productions of some grammar;
- any theorem derived in some logic system is obtained by applying some axioms and inference rules to form a (finite) derivation tree;
- any computation is obtained by composing single steps.

If we want to prove non-trivial properties of a program or of a class of programs, we usually have to use *induction* principles. The most general notion of induction is the so-called *well-founded induction* (or *Noetherian* induction) and we derive from it all the other induction principles.

In the above cases (arbitrarily large but finitely generated elements) we can exploit the induction principle to prove a universally quantified statement by showing that

base case: the statement holds in all possible elementary cases (e.g., 0, the sentences of the grammar obtained by applying productions involving non-terminal symbols only, the basic derivations of a proof system obtained by applying the axioms, the initial step of a computation);

inductive case: and that the statement holds in the composite cases (e.g. $succ(n)$, the terms of the grammar obtained by applying productions involving non-terminal symbols, the derivations of a proof system

obtained by applying an inference rule to smaller derivations, a computation of $n+1$ steps, etc.), under the assumption that it holds in any simpler case (e.g., for any $k \leq n$, for any subterm, for any smaller derivation, for any computation whose length is less than or equal to n).

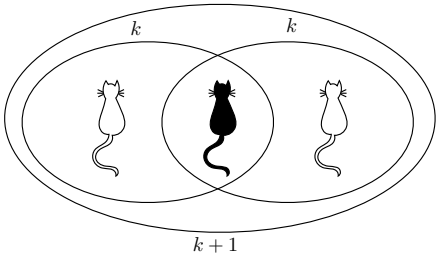
Exercise 1.3. Induction can be deceptive. Let us consider the following argument for proving that all cats are the same colour.

Let $P(n)$ be the proposition that: *In a group of n cats, all cats are the same colour*

The statement is trivially true for $n = 1$ (base case).

For the inductive case, suppose that the statement is true for $n \leq k$. Take a group of $k+1$ cats: we want to prove that they are the same colour.

Align the cats along a line. Form two groups of k cats each: the first k cats in the line and the last k cats in the line. By the inductive hypothesis, the cats in each of the two groups are the same colour. Since the cat in the middle of the line belongs to both groups, by transitivity all cats in the line are the same colour. Hence $P(k+1)$ is true.



By induction, $P(n)$ is true for all $n \in \mathbb{N}$. Hence, all cats are the same colour.

We know that this cannot be the case: what's wrong with the above reasoning?

The usual proof technique for proving properties of a natural semantics definition is induction on the height of the derivation trees that are generated from the semantics, from which we get the special case of rule induction:

base cases:	P holds for each axiom, and
inductive cases:	for each inference rule, if P holds for the premises, then it holds for the conclusion.

For proving properties of a denotational semantics, induction on the structure of the terms is often a convenient proof strategy.

Defining the denotational semantics of a composed program by *structural recursion* means assigning its meaning in terms of the meanings of its components. We will see that induction and recursion are very similar: for both induction and recursion we will need well-founded models.

1.4.2 Semantic Domains

The choice of a suitable semantic domain is not always as easy as in the example of numerical expressions.

For example, the semantics of programs is often formulated in a functional space, from the domain of states to itself (i.e., a program is seen as a state transformation). The functions we need to consider can be partial ones, if the programs can diverge. Note that the domain of states can also be a complex structure, e.g., a state can be an assignment of values to variables.

If we take a program which is cyclic or recursive, then we have to express these notions at the level of the meanings, which presents some technical difficulties.

A recursive program p contains a call to itself, therefore to assign a meaning $[\![p]\!]$ to the program p we need to solve a recursive equation of the form

$$[\![p]\!] = f([\![p]\!]). \tag{1.1}$$

In general, it can happen than such equations have zero, one or many solutions. Solutions to recursive equations are called *fixpoints*.

Example 1.3. Let us consider the domain of natural numbers:

$n = 2 \times n$	has only one solution: $n = 0$
$n = n + 1$	has no solution
$n = 1 \times n$	has many solutions: any n

Example 1.4. Let us consider the domain of sets of integers:

$X = X \cap \{1\}$	has two solutions: $X = \varnothing$ or $X = \{1\}$
$X = \mathbb{N} \setminus X$	has no solution
$X = X \cup \{1\}$	has many solutions: any $X \supseteq \{1\}$

In order to provide a general solution to this kind of problem, we resort to the theory of *complete partial orders with bottom* and of *continuous* functions.

In the functional programming paradigm, a higher-order functional language can use functions as arguments to other functions, i.e., spaces of functions must also be considered as forming data types. This makes the language's domains more complex. Denotational semantics can be used to understand these complexities; an applied branch of mathematics called *domain theory* is used to formalise the domains with algebraic equations.

Let us consider a domain D where we interpret the elements of some data type. The idea is that two elements $x, y \in D$ are not necessarily separated, but one, say y, can be a better version of what x is trying to approximate, written

$$x \sqsubseteq y$$

with the intuition that y is *consistent* with x and is (possibly) *more accurate* than x.

Concretely, an especially interesting case is when one can take two partial functions f, g and say that g is a better approximation than f if whenever $f(x)$ is defined then also $g(x)$ is defined and $g(x) = f(x)$. But g can be defined on elements on which f is not.

Note that if we see (partial) functions as relations (sets of pairs $(x, f(x))$), then the above concept boils down to set inclusion.

For example, we can progressively approximate the factorial function by taking the sequence of partial functions

$$\varnothing \subseteq \{(1,1)\} \subseteq \{(1,1),(2,2)\} \subseteq \{(1,1),(2,2),(3,6)\} \subseteq \{(1,1),(2,2),(3,6),(4,24)\} \subseteq \cdots$$

Now, it is quite natural to require that our notion of approximation $\sqsubseteq$ is reflexive, transitive and antisymmetric: this means that our domain D is a *partial order*.

Often there is an element, called *bottom* and denoted by $\bot$, which is less defined than any other element: in our example about partial functions, the bottom element is the partial function $\varnothing$.

When we apply a function f (determined by some program) to elements of D it is also quite natural to require that the more accurate the input, the more accurate the result:

$$x \sqsubseteq y \quad \Rightarrow \quad f(x) \sqsubseteq f(y)$$

This means that our functions of interest are *monotonic*.

Now suppose we are given an infinite sequence of approximations

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$$

It seems reasonable to suppose that the sequence tends to some limit that we denote as $\bigsqcup_n x_n$ and moreover that mappings between data types are well behaved w.r.t. limits, i.e., that data transformations are *continuous*:

$$f\left(\bigsqcup_n x_n\right) = \bigsqcup_n f(x_n)$$

Interestingly, one can prove that for a function to be continuous in several variables jointly, it is sufficient that it be continuous in each of its variables separately.

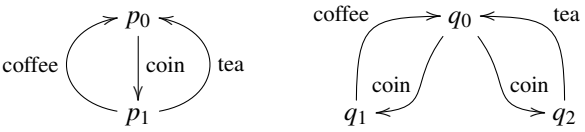
Kleene's fixpoint theorem ensures that when continuous functions are considered over complete partial orders (with bottom) then a suitable *least* fixpoint exists, and tells us how to compute it. The fixpoint theory is first applied to the case of IMP semantics and then extended to handle HOFL. The case of HOFL is more complex because we are working on a more general situation where functions are first-class citizens.

When defining coarsest equivalences over concurrent processes, we also present a weaker version of the fixpoint theorem by Knaster and Tarski that can be applied to monotone functions (not necessarily continuous) over complete lattices.

1.4.3 Bisimulation

The models we use for IMP and HOFL are not appropriate for concurrent and interactive systems, such as the very common network-based applications: on the one hand we want their behaviour to depend as little as possible on the speed of processes, on the other hand we want to permit infinite computations and to differentiate among them on the basis of the interactions they can undertake. For example, in the case of IMP and HOFL all diverging programs are considered equivalent. The typical models for nondeterminism and infinite computations are *(labelled) transition systems*. We do not consider time explicitly, but we have to introduce nondeterminism to account for races between concurrent processes.

In the case of interactive, concurrent systems, as represented by labelled transition systems, the classic notion of language equivalence from finite automata theory is not well suited as a criterion for program equivalence, because it does not account properly for non-terminating computations and nondeterministic behaviour. To see this, consider the two labelled transition systems below, which model the behaviour of two different coffee machines:



It is evident that any sequence of actions that is executable by the first machine can be also executed by the second machine, and vice versa. However, from the point of view of the interaction with the user, the two machines behave very differently: after the introduction of the coin, the machine on the left still allows the user to choose between a coffee and a tea, while the machine on the right leaves no choice to the user.

We show that a suitable notion of equivalence between concurrent, interactive systems can be defined as a *behavioural equivalence* called *bisimulation*: it takes into account the branching structure of labelled transition systems as well as infinite computations. Equivalent programs are represented by (initial) states which have corresponding observable transitions leading to equivalent states. Interestingly, there is a nice connection between fixpoint theory and the definition of the coarsest bisimulation equivalence, called *bisimilarity*. Moreover, bisimilarity finds a logical counterpart in *Hennessy-Milner logic*, in the sense that two systems are bisimilar if and only if they satisfy the same Hennessy-Milner logic formulas. Beside using bisimilarity to compare different realisations of the same system, weaker forms of bisimilarity can be used to study the compliance between an abstract specification and a concrete implementation.

The language that we employ in this setting is a *process algebra* called CCS (*Calculus of Communicating Systems*). Then, we study systems whose communication structure can change during execution. These systems are called *open-ended*. As our case study, we present the π-calculus, which extends CCS. The π-calculus

is quite expressive, due to its ability to create and transmit new names, which can represent ports, links and also session names, nonces, passwords and so on in security applications.

1.4.4 Temporal and Modal Logics

We investigate also *temporal* and *modal logics* designed to conveniently express properties of concurrent, interactive systems.

Modal logics were conceived in philosophy to study different *modes of truth*, for example an assertion being false in the current world but *possibly* true in some alternate world, or another holding *always* true in all worlds. Temporal logics are an instance of modal logics for reasoning about the truth of assertions over time. Typical temporal operators includes the possibility to assert that a property is true *sometimes* in the future, or that it is *always* true, in all future moments. The most popular temporal logics are LTL (Linear Temporal Logic) and CTL (Computation Tree Logic). They have been extensively studied and used for applying formal methods to industrial case studies and for the specification and verification of program correctness.

We introduce the basics of LTL and CTL and then present a modal logic with recursion, called the μ*-calculus*, that encompasses LTL and CTL. The definition of the semantics of the μ-calculus exploits again the principles of domain theory and fixpoint computation.

1.4.5 Probabilistic Systems

Finally, in the last part of the book we focus on probabilistic models, where we trade nondeterminism for probability distributions, which we associate with choice points.

Probability theory plays a big role in modern computer science. It focuses on the study of random events, which are central in areas such as artificial intelligence and network theory, e.g., to model variability in the arrival of requests and predict load distribution on servers. Probabilistic models of computation assign weight to choices and refine nondeterministic choices with probability distributions. In interactive systems, when many actions are enabled at the same time, the probability distribution models the frequency with which each alternative can be executed. Probabilistic models can also be used in conjunction with sources of nondeterminism and we present several ways in which this can be achieved. We also present stochastic models, where actions take place in a continuous time setting, with an exponential distribution.

A compelling case of probabilistic systems is given by *Markov chains*, which represent random processes over time. We study two kinds of Markov chains, which differ in the way in which time is represented (discrete vs continuous) and we focus

on homogeneous chains only, where the distribution depends on the current state of the system, but not on the current time. For example, in some special cases, Markov chains can be used to estimate the probability of finding the system in a given state in the long run or the probability that the system will not change its state in some time.

By analogy with labelled transition systems we are also able to define suitable notions of bisimulation and an analogue of Hennessy-Milner logic, called *Larsen-Skou logic*. Finally, by analogy with CCS, we present a high-level language for the description of continuous time Markov chains, called PEPA, which can be used to define stochastic systems in a structured and compositional way as well as by refinement from specifications. PEPA is tool supported and has been successfully applied in many fields, e.g., performance evaluation, decision support systems and system biology.

1.5 Chapter Contents and Reading Guide

After Chapter 2, where some notation is fixed and useful preliminaries about logical systems, goal-oriented derivations and proof strategies are explained, the book comprises four main parts: the first two parts exemplify deterministic systems; the other two model nondeterministic ones. The difference will become clear during reading.

- Computational models for imperative languages, exemplified by IMP:
  - In Chapter 3 the simple imperative language IMP is introduced; its natural semantics is defined and studied together with the induced notion of program equivalence.
  - In Chapter 4 the general principle of well-founded induction is stated and related to other widely used induction principles, such as mathematical induction, structural induction and rule induction. The chapter is concluded with an illustration of well-founded recursive definitions.
  - In Chapter 5 the mathematical basis for denotational semantics is presented, including the concepts and properties of complete partial orders, least upper bounds and monotone and continuous functions. In particular this chapter contains Kleene's fixpoint theorem, which is used extensively in the rest of the monograph, and the definition of the immediate consequence operator associated with a logical system, which is exploited in Chapter 6. The presentation of Knaster-Tarski's fixpoint is instead postponed to Chapter 11.
  - In Chapter 6 the foundations introduced in Chapter 5 are exploited to define the denotational semantics of IMP and to derive a corresponding notion of program equivalence. The induction principles studied in Chapter 4 are then exploited to prove the correspondence between the operational and denotational semantics of IMP and consequently between their two induced equivalences over processes. The chapter is concluded with the presentation of Scott's principle of *computational induction* for proving *inclusive* properties.
- Computational models for functional languages, exemplified by HOFL:

  - In Chapter 7 we shift from the imperative style of programming to the declarative one. After presenting the λ-notation, useful for representing anonymous functions, the higher-order functional language HOFL is introduced, where infinitely many data types can be constructed by pairing and function type constructors. Church type theory and Curry type theory are discussed and the unification algorithm from Chapter 2 is used for type inference. Typed terms are given a natural semantics called *lazy*, because it evaluates a parameter of a function only if needed. The alternative *eager* semantics, where actual arguments are always evaluated, is also discussed.
  - In Chapter 8 we extend the theory presented in Chapter 5 to allow the construction of more complex domains, as needed by the type constructors available in HOFL.
  - In Chapter 9 the foundations introduced in Chapter 8 are exploited to define the (lazy) denotational semantics of HOFL.
  - In Chapter 10 the operational and denotational semantics of HOFL are compared, by showing that the notion of program equivalence induced by the former is generally stricter than the one induced by the latter and that they coincide only over terms of type integer. However, it is shown that the two semantics are equivalent w.r.t. the notion of convergence.

- Computational models for concurrent/nondeterministic/interactive languages, exemplified by CCS and the π-calculus:

  - In Chapter 11 we shift the focus from sequential systems to concurrent and interactive ones. The process algebra CCS is introduced which allows the description of concurrent communicating systems. Such systems communicate by message passing over named channels. Their operational semantics is defined in the small-step style, because infinite computations must be accounted for. Communicating processes are assigned labelled transition systems by inference rules in the SOS-style and several equivalences over such transition systems are discussed. In particular the notion of behavioural equivalence is put forward, in the form of bisimulation equivalence. Notably, the coarsest bisimulation, called bisimilarity, exists, it can be characterised as a fixpoint, it is a congruence w.r.t. the operators of CCS and it can be axiomatised. Its logical counterpart, called Hennessy-Milner logic, is also presented. Finally, coarser equivalences are discussed, which can be exploited to relate system specifications with more concrete implementations by abstracting away from internal moves.
  - In Chapter 12 some logics are considered that increase the expressiveness of Hennessy-Milner logic by defining properties of finite and infinite computations. First the temporal logics LTL and CTL are presented, and then the more expressive μ-calculus is studied. The notion of satisfaction for μ-calculus formulas is defined by exploiting fixpoint theory.
  - In Chapter 13 the theory of concurrent systems is extended with the possibility to communicate channel names and create new channels. Correspondingly,

we move from CCS to the π-calculus, we define its small-step operational semantics and we introduce several notions of bisimulation equivalence.

- Computational models for probabilistic and stochastic process calculi:
  - In Chapter 14 we shift the focus from nondeterministic systems to probabilistic ones. After introducing the basics of measure theory and the notions of random process and Markov property, two classes of random processes are studied, which differ in the way time is represented: DTMC (discrete time) and CTMC (continuous time). In both cases, it is studied how to compute a stationary probability distribution over the possible states and a suitable notion of bisimulation equivalence.
  - In Chapter 15, the various possibilities for defining probabilistic models of computation with observable actions and sources of nondeterminism are overviewed, emphasising the difference between reactive models and generative ones. Finally a probabilistic version of Hennessy-Milner logic is presented, called Larsen-Skou logic.
  - In Chapter 16 a well-known high-level language for the specification and analysis of stochastic interactive systems, called PEPA (Performance Evaluation Process Algebra), is presented. The small-step operational semantics of PEPA is first defined and then it is shown how to associate a CTMC with each PEPA process.

1.6 Further Reading

One leitmotif of this monograph is the use of logical systems of inference rules. As derivation trees tend to grow large very fast, even for small examples, we will introduce and rely on goal-oriented derivations inspired by logic programming, as explained in Section 2.3. A nice introduction to the subject can be found in the lecture notes[8] by Frank Pfenning [26]. The first chapters cover, in a concise but clear manner, most of the concepts we shall exploit.

The reader interested in knowing more about the theory of partial orders and domains is referred to (the revised edition of) the book by Davey and Priestley [5] for a gentle introduction to the basic concepts and to the chapter by Abramsky and Jung in the Handbook of Logic in Computer Science for a full account of the subject [1]. A freely available document on domain theory that accounts also for the case of parallel and nondeterministic systems is the so-called "Pisa notes" by Plotkin [29]. The reader interested in denotational semantics methods only can then consult [35] for an introduction to the subject and [10] for a comprehensive treatment of programming language constructs, including different procedure call mechanisms.

There are several books on the semantics of imperative and functional languages [12, 41, 23, 24, 32, 40, 7]. For many years, we have adopted the book

[8] Freely available at the time of publication.

by Glynn Winskel [43] for the courses in Pisa, which is possibly the closest to our approach. It covers most of the content of Parts II (IMP) and III (HOFL) and has a chapter on CCS and modal logic (see Part IV), there discussed together with another well-known process algebra for concurrent systems called CSP (for Communicating Sequential Processes) and introduced by Tony Hoare. The main differences are that we use goal-oriented derivations for logical systems (type systems and operational semantics) and focus on the lazy semantics of HOFL, while Winskel's book exploits derivation trees and gives a detailed treatment of the eager semantics of HOFL. We briefly discuss CSP in Chapter 16 as it is the basis for PEPA. Chapter 13 and Part V are not covered in [43]. We briefly overview elementary type systems in connection to HOFL. To deepen the study of the topic, including polymorphic and recursive types, we recommend the books by Benjamin Pierce [27] and by John Mitchell [23].

Moving to Part IV, the main and most-cited reference for CCS is Robin Milner's book [20]. However, for an up-to-date presentation of CCS theory, we refer to the very recent book by Roberto Gorrieri and Cristian Versari [11]. Both texts are complementary to the book by Luca Aceto et al. [2], where the material is organised to place the emphasis on verification aspects of concurrent systems and CCS is presented as a useful formal tool. Mobility is not considered in the above texts. The basic reference for the π-calculus is the seminal book by Robin Milner [21]. The whole body of theory that has been subsequently developed is presented at a good level of detail in the book by Davide Sangiorgi and David Walker [34]. Many free tutorials on CCS and the π-calculus can also be found on the web.

Bisimulation equivalences are presented and exploited in all the above books, but their use, as well as that of the more general concept of coinduction, spans far beyond CCS and interactive systems. The new book by Davide Sangiorgi [33] explores the subject from many angles and provides good insights. The algorithmic-minded reader is also referred to the recent survey [3].

The literature on temporal and modal logics and their applications to verification and model checking is quite vast and falls outside the scope of our book. We just point the reader to the compact survey on the modal μ-calculus by Giacomo Lenzi [18], which explains synthetically how LTL and CTL can be seen as sublogics of the μ-calculus, and to the book by Christel Baier and Joost-Pieter Katoen on model checking principles [4], where also verification of probabilistic systems is addressed.

This brings us to Part V. We think one peculiarity of this monograph is that it groups under the same umbrella several paradigms that are often treated in separation. This is certainly the case with Markov chains and probabilistic systems. Markov chains are usually studied in first courses on probability for Computer Science. Their combined use with transitions for interaction is a more advanced subject and we refer the interested reader to the well-known book by Prakash Panangaden [25].

Finally, PEPA, where the process algebraic approach merges with the representation of stochastic systems, allowing modelling and measurement of not just the expressiveness of processes but also their performance, from many angles. The introductory text for PEPA principles is the book by Jane Hillston [14] possibly accompanied by the short presentation in [13]. For people interested in experimenting with PEPA we refer instead to [9].

References

1. Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, Oxford, 1994.
2. Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jirí Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, 2007.
3. Luca Aceto, Anna Ingólfsdóttir, and Jirí Srba. The algorithmics of bisimilarity. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, pages 100–172. Cambridge University Press, 2011. Cambridge Books Online.
4. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, 2008.
5. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 2002.
6. Véronique Donzeau-Gouge, Gilles Kahn, and Bernard Lang. On the formal definition of ADA. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980*, volume 94 of *Lecture Notes in Computer Science*, pages 475–489. Springer, Berlin, 1980.
7. Maribel Fernández. *Programming Languages and Operational Semantics - A Concise Overview*. Undergraduate Topics in Computer Science. Springer, Berlin, 2014.
8. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, 1967.
9. Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In Günter Haring and Gabriele Kotsis, editors, *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference, Vienna, Austria, May 3-6, 1994, Proceedings*, volume 794 of *Lecture Notes in Computer Science*, pages 353–368. Springer, Berlin, 1994.
10. M.J.C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer, Berlin, 1979.
11. Roberto Gorrieri and Cristian Versari. *Introduction to Concurrency Theory - Transition Systems and CCS*. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin, 2015.
12. Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, New York, 1990.
13. Jane Hillston. Compositional Markovian modelling using a process algebra. In William J. Stewart, editor, *Computations with Markov Chains: Proceedings of the 2nd International Workshop on the Numerical Solution of Markov Chains*, pages 177–196. Springer, Berlin, 1995.
14. Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, 1996.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
16. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, Berlin, 1987.
17. Richard Kelsey, William D. Clinger, and Jonathan Rees, editors. Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.
18. Giacomo Lenzi. The modal μ-calculus: a survey. *TASK Quarterly*, 9(3):293–316, 2005.
19. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.
20. Robin Milner. *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, Upper Saddle River, 1989.
21. Robin Milner. *Communicating and Mobile Systems - the π-calculus*. Cambridge University Press, New York, 1999.

22. Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, Cambridge, 1990.
23. John C. Mitchell. *Foundations for Programming Languages*. Foundation of Computing Series. MIT Press, Cambridge, 1996.
24. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications - a Formal Introduction*. Wiley Professional Computing. Wiley, New York, 1992.
25. Prakash Panangaden. *Labelled Markov Processes*. Imperial College Press, London, 2009.
26. Frank Pfenning. Logic programming, 2007. Lecture notes. Available at `http://www.cs.cmu.edu/~fp/courses/lp/`.
27. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, 2002.
28. Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus, Denmark, 1981.
29. Gordon D. Plotkin. Domains (the 'Pisa' Notes), 1983. Notes for lectures at the University of Edinburgh, extending lecture notes for the Pisa summer school 1978. Available at `http://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps`.
30. Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.
31. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
32. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, 1998.
33. Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, 2011.
34. Davide Sangiorgi and David Walker. *The π-calculus - a Theory of Mobile Processes*. Cambridge University Press, Cambridge, 2001.
35. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown, Dubuque, 1986.
36. David A. Schmidt. Programming language semantics. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2237–2254. CRC Press, Boca Raton, 1997.
37. Dana S. Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG–2, Programming Research Group, Oxford, November 1970.
38. Dana S. Scott. Some reflections on Strachey and his work. *Higher-Order and Symbolic Computation*, 13(1/2):103–114, 2000.
39. Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46. Polytechnic Press, Brooklyn, 1971.
40. Aaron Stump. *Programming Language Foundations*. Wiley, New York, 2013.
41. Robert D. Tennent. *Semantics of Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall, Upper Saddle River, 1991.
42. Adriaan van Wijngaarden, B. J. Mailloux, J. E. L. Peck, Cornelis H. A. Koster, Michel Sintzoff, C. H. Lindsey, L. G. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Inf.*, 5(1):1–236, 1975.
43. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, Cambridge, 1993.

Chapter 2
Preliminaries

A mathematician is a device for turning coffee into theorems.
(Paul Erdös)

Abstract In this chapter we fix some basic mathematical notation used in the rest of the book and introduce the important concepts of signature, logical system and goal-oriented derivation.

2.1 Notation

2.1.1 Basic Notation

As a general rule, we use capital letters, like D or X, to denote sets of elements, and small letters, like d or x, for their elements, with membership relation $\in$. The set of natural numbers is denoted by $\mathbb{N} \stackrel{\text{def}}{=} \{0,1,2,\ldots\}$, the set of integer numbers is denoted by $\mathbb{Z} \stackrel{\text{def}}{=} \{\ldots,-2,-1,0,1,2,\ldots\}$ and the set of booleans by $\mathbb{B} \stackrel{\text{def}}{=} \{\mathbf{true},\mathbf{false}\}$. We write $[m,n] \stackrel{\text{def}}{=} \{k \mid m \leq k \leq n\}$ for the interval of numbers from m to n, extremes included. If a set A is finite, we denote by $|A|$ its *cardinality,* i.e., the number of its elements. The empty set is written $\varnothing$, with $|\varnothing| = 0$. We use the standard notation for set union, intersection, difference, cartesian product and disjoint union, which are denoted respectively by $\cup$, $\cap$, $\setminus$, $\times$ and $\uplus$. We write $A \subseteq B$ if all elements in A belong to B. We denote by $\wp(A)$ the powerset of A, i.e., the set of all subsets of A.

An indexed set of elements is written $\{e_i\}_{i\in I}$ and a family of sets is written $\{S_i\}_{i\in I}$. Set operations are extended to families of sets by writing, e.g., $\bigcup_{i\in I} S_i$ and $\bigcap_{i\in I} S_i$. If I is the interval set $[m,n]$, then we write also $\bigcup_{i=m}^{n} S_i$ and $\bigcap_{i=m}^{n} S_i$.

Given a set A and a natural number k we denote by A^k the set of sequences of k (not necessarily distinct) elements in A. Such sequences are called *strings* and their concatenation is represented by juxtaposition. We denote by $A^* = \bigcup_{k\in\mathbb{N}} A^k$ the set of all finite (possibly empty) sequences over A. Given a string $w \in A^*$ we denote by $|w|$ its *length*, i.e., the number of its elements (including repetitions). The empty string is denoted ε, and we have $|\varepsilon| = 0$ and $A^0 = \{\varepsilon\}$ for any A. We denote by $A^+ = \bigcup_{k>0} A^k = A^* \setminus \{\varepsilon\}$ the set of all finite nonempty sequences over A.

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_2

A relation R between two sets A and B is a subset of $A \times B$. For $(a,b) \in R$ we write also aRb. A relation $f \subseteq A \times B$ can be regarded as a function if both the following properties are satisfied:

function: $\forall a \in A, \forall b_1, b_2 \in B.\ (a,b_1) \in f \wedge (a,b_2) \in f \ \Rightarrow\ b_1 = b_2$
total: $\forall a \in A, \exists b \in B.\ (a,b) \in f$

For such a function f, we write $f : A \to B$ and say that the set A is the *domain* of f, and B is its *codomain*. We write $f(a)$ for the unique element $b \in B$ such that $(a,b) \in f$, i.e., f can be regarded as the relation $\{(a, f(a)) \mid a \in A\} \subseteq A \times B$. The composition of two functions $f : A \to B$ and $g : B \to C$ is written $g \circ f : A \to C$, and it is such that for any element $a \in A$ it holds that $(g \circ f)(a) = g(f(a))$. A relation that satisfies the "function" property, but not necessarily the "total" property, is called *partial*. A partial function f from A to B is written $f : A \rightharpoonup B$.

2.1.2 Signatures and Terms

A one-sorted (or unsorted) *signature* is a set of function symbols $\Sigma = \{c, f, g, \ldots\}$ such that each symbol in Σ is assigned an *arity*, that is the number of arguments it takes. A symbol with arity zero is called a *constant*; a symbol with arity one is called *unary*; a symbol with arity two is called *binary*; a symbol with arity three is called *ternary*. For $n \in \mathbb{N}$, we let $\Sigma_n \subseteq \Sigma$ be the set of function symbols whose arity is n.

Given an infinite set of variables $X = \{x, y, z, \ldots\}$, the set $T_{\Sigma,X}$ is the set of terms over Σ and X, i.e., the set of all and only terms generated according to the following rules:

- each variable $x \in X$ is a term (i.e., $x \in T_{\Sigma,X}$),
- each constant $c \in \Sigma_0$ is a term (i.e., $c \in T_{\Sigma,X}$),
- if $f \in \Sigma_n$, and $t_1, \ldots, t_n$ are terms (i.e., $t_1, \ldots, t_n \in T_{\Sigma,X}$) then also $f(t_1, \ldots, t_n)$ is a term (i.e., $f(t_1, \ldots, t_n) \in T_{\Sigma,X}$).

For a term $t \in T_{\Sigma,X}$, we denote by $\mathrm{vars}(t)$ the set of variables occurring in t, and let $T_\Sigma \subseteq T_{\Sigma,X}$ be the set of terms with no variables, i.e., $T_\Sigma \stackrel{\text{def}}{=} \{t \in T_{\Sigma,X} \mid \mathrm{vars}(t) = \varnothing\}$.

Example 2.1. Take $\Sigma = \{0, succ, plus\}$ with 0 a constant, *succ* unary and *plus* binary. Then all of the following are terms:

- $0 \in T_\Sigma$
- $x \in T_{\Sigma,X}$
- $succ(0) \in T_\Sigma$
- $succ(x) \in T_{\Sigma,X}$
- $plus(succ(x), 0) \in T_{\Sigma,X}$
- $plus(plus(x, succ(y)), plus(0, succ(x))) \in T_{\Sigma,X}$

The set of variables of the above terms are respectively

- $\mathrm{vars}(\,0\,) = \mathrm{vars}(\,succ(0)\,) = \varnothing$

- $\mathrm{vars}(x) = \mathrm{vars}(\mathit{succ}(x)) = \mathrm{vars}(\mathit{plus}(\mathit{succ}(x),0)) = \{x\}$
- $\mathrm{vars}(\mathit{plus}(\mathit{plus}(x,\mathit{succ}(y)),\mathit{plus}(0,\mathit{succ}(x)))) = \{x,y\}$

Instead $\mathit{succ}(\mathit{plus}(0),x)$ is not a term: can you see why?

2.1.3 Substitutions

A *substitution* $\rho : X \to T_{\Sigma,X}$ is a function assigning terms to variables.

Since the set of variables is infinite while we are interested only in terms with a finite number of variables, we consider only substitutions that are defined as identity everywhere except on a finite number of variables. Such substitutions are written

$$\rho = [x_1 = t_1 , \ldots , x_n = t_n]$$

meaning that

$$\rho(x) = \begin{cases} t_i & \text{if } x = x_i \\ x & \text{otherwise} \end{cases}$$

We denote by $t\rho$, or sometimes by $\rho(t)$, the term obtained from t by simultaneously replacing each variable x with $\rho(x)$ in t.

Example 2.2. For example, consider the signature from Example 2.1, the term $t \stackrel{\text{def}}{=} \mathit{plus}(\mathit{succ}(x),\mathit{succ}(y))$ and the substitution $\rho \stackrel{\text{def}}{=} [x = \mathit{succ}(y) , y = 0]$. We get:

$$t\rho = \mathit{plus}(\mathit{succ}(x),\mathit{succ}(y))[x = \mathit{succ}(y) , y = 0] = \mathit{plus}(\mathit{succ}(\mathit{succ}(y)),\mathit{succ}(0))$$

We say that the term t is *more general* than the term t' if there exists a substitution ρ such that $t\rho = t'$. The "more general than" relation is reflexive and transitive, i.e., it defines a *pre-order*. Note that there are terms t and t', with $t \neq t'$, such that t is more general than t' and t' is more general than t.

We say that the substitution ρ is more general than the substitution ρ' if there exists a substitution ρ'' such that for any variable x we have that $\rho''(\rho(x)) = \rho'(x)$ (i.e., $\rho(x)$ is more general than $\rho'(x)$ as witnessed by ρ'').

2.1.4 Unification Problem

The unification problem, in its simplest formulation (syntactic, first-order unification), consists of finding a substitution ρ that identifies some terms pairwise.

Formally, given a set of potential equalities

$$G = \{l_1 \stackrel{?}{=} r_1, \ldots, l_n \stackrel{?}{=} r_n\}$$

where $l_i, r_i \in T_{\Sigma,X}$, we say that a substitution ρ is a solution of G if

$$\forall i \in [1,n].\ l_i\rho = r_i\rho$$

The unification problem requires to find a *most general* solution ρ.

We say that two sets of potential equalities G and G' are *equivalent* if they have the same set of solutions.

We denote by $\mathrm{vars}(G)$ the set of variables occurring in G, i.e.,

$$\mathrm{vars}(\{l_1 \stackrel{?}{=} r_1, \ldots, l_n \stackrel{?}{=} r_n\}) = \bigcup_{i=1}^{n} (\mathrm{vars}(l_i) \cup \mathrm{vars}(r_i))$$

Note that the solution does not necessarily exist, and when it exists it is not necessarily unique.

The unification algorithm takes as input a set of potential equalities G like the one above and applies some transformations until

- either it terminates (no transformation can be applied any more) after having transformed the set G into an equivalent set of equalities

  $$G' = \{x_1 \stackrel{?}{=} t_1, \ldots, x_k \stackrel{?}{=} t_k\}$$

  where $x_1, \ldots, x_k$ are all distinct variables and $t_1, \ldots, t_k$ are terms with no occurrences of $x_1, \ldots, x_k$, i.e., such that $\{x_1, \ldots, x_k\} \cap \bigcup_{i=1}^{k} \mathrm{vars}(t_i) = \varnothing$: the set G' directly defines a most general solution

  $$[x_1 = t_1, \ldots, x_k = t_k]$$

  to the unification problem G;
- or it fails, meaning that the potential equalities cannot be unified.

In the following we denote by $G\rho$ the set of potential equalities obtained by applying the substitution ρ to all terms in G. Formally

$$\{l_1 \stackrel{?}{=} r_1, \ldots, l_n \stackrel{?}{=} r_n\}\rho = \{l_1\rho \stackrel{?}{=} r_1\rho, \ldots, l_n\rho \stackrel{?}{=} r_n\rho\}$$

The unification algorithm tries to apply the following steps (the order is not important for the result, but it may affect complexity), to transform an initial set of potential equalities until no more steps can be applied or the algorithm fails:

delete:	$G \cup \{t \stackrel{?}{=} t\}$ is transformed into G
decompose:	$G \cup \{f(t_1, \ldots, t_m) \stackrel{?}{=} f(u_1, \ldots, u_m)\}$ is transformed into $G \cup \{t_1 \stackrel{?}{=} u_1, \ldots, t_m \stackrel{?}{=} u_m\}$
swap:	$G \cup \{f(t_1, \ldots, t_m) \stackrel{?}{=} x\}$ is transformed into $G \cup \{x \stackrel{?}{=} f(t_1, \ldots, t_m)\}$
eliminate:	$G \cup \{x \stackrel{?}{=} t\}$ is transformed into $G[x = t] \cup \{x \stackrel{?}{=} t\}$ if $x \in \mathrm{vars}(G) \wedge x \notin \mathrm{vars}(t)$
conflict:	$G \cup \{f(t_1, \ldots, t_m) \stackrel{?}{=} g(u_1, \ldots, u_h)\}$ leads to failure if $f \neq g \vee m \neq h$
occur check:	$G \cup \{x \stackrel{?}{=} f(t_1, \ldots, t_m)\}$ leads to failure if $x \in \mathrm{vars}(f(t_1, \ldots, t_m))$

Example 2.3. For example, if we start from

$$G = \{plus(succ(x), x) \stackrel{?}{=} plus(y, 0)\}$$

by applying rule **decompose** we obtain

$$\{succ(x) \stackrel{?}{=} y\,,\, x \stackrel{?}{=} 0\}$$

by applying rule **eliminate** we obtain

$$\{succ(0) \stackrel{?}{=} y\,,\, x \stackrel{?}{=} 0\}$$

finally, by applying rule **swap** we obtain

$$\{y \stackrel{?}{=} succ(0)\,,\, x \stackrel{?}{=} 0\}$$

Since no further transformation is possible, we conclude that

$$\rho = [y = succ(0)\,,\, x = 0]$$

is the most general unifier for G.

2.2 Inference Rules and Logical Systems

Inference rules are a key tool for defining syntax (e.g., which programs respect the syntax, which programs are well typed) and semantics (e.g., to derive the operational semantics by induction on the syntactic structure of programs).

Definition 2.1 (Inference rule). Let $x_1, x_2, \ldots, x_n, y$ be (well-formed) formulas. An *inference rule* is written, using inline notation, as

$$r \quad = \quad \underbrace{\{x_1, x_2, \ldots, x_n\}}_{\text{premises}} \quad / \quad \underbrace{y}_{\text{conclusion}}$$

Letting $X = \{x_1, x_2, \ldots, x_n\}$, equivalent notations are

$$r = \frac{X}{y} \qquad\qquad r = \frac{x_1 \quad \ldots \quad x_n}{y}$$

The meaning of such a rule r is that if we can prove all the formulas $x_1, x_2, \ldots, x_n$ in our logical system, then by exploiting the inference rule r we can also derive the validity of the formula y.

Definition 2.2 (Axiom). An *axiom* is an inference rule with empty premise:

$$r = \varnothing / y.$$

Equivalent notations are

$$r = \frac{\varnothing}{y} \qquad\qquad r = \frac{}{y}$$

In other words, there are no pre-conditions for applying an axiom r, hence there is nothing to prove in order to apply the rule: in this case we can assume y to hold.

Definition 2.3 (Logical system). A *logical system* is a set of inference rules $R = \{r_i\}_{i \in I}$.

Given a logical system, we can start by deriving obvious facts using axioms and then derive new valid formulas by applying the inference rules to the formulas that we know to hold (used as premises). In turn, the newly derived formulas can be used to prove the validity of other formulas.

Example 2.4 (Some inference rules). The inference rule

$$\frac{x \in E \quad y \in E \quad x \oplus y = z}{z \in E}$$

means that, *if* x and y are two elements that belong to the set E *and* the result of applying the operator $\oplus$ to x and y gives z as a result, *then* z must also belong to the set E.

The rule

$$\frac{}{2 \in E}$$

is an axiom, so we know that 2 belongs to the set E.

By composing inference rules, we build *derivations*, which explain how a logical deduction is achieved.

Definition 2.4 (Derivation). Given a logical system R, a *derivation* is written

$$d \Vdash_R y$$

where

- either $d = \varnothing / y$ is an axiom of R, i.e., $(\varnothing / y) \in R$;
- or there are some derivations $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ such that $d = (\{d_1, \ldots, d_n\} / y)$ and $(\{x_1, \ldots, x_n\} / y) \in R$.

The notion of derivation is obtained by putting together different steps of reasoning according to the rules in R. We can see $d \Vdash_R y$ as a proof that, in the formal system R, we can derive y.

Let us look more closely at the two cases in Definition 2.4. The first case tells us that if we know that

$$\left(\frac{\varnothing}{y}\right) \in R$$

i.e., if we have an axiom for deriving y in our inference system R, then

$$\left(\frac{\varnothing}{y}\right) \Vdash_R y$$

is a derivation of y in R.

The second case tells us that if we have already proved x_1 with derivation d_1, x_2 with derivation d_2 and so on, i.e.,

$$d_1 \Vdash_R x_1, \qquad d_2 \Vdash_R x_2, \qquad \ldots, \qquad d_n \Vdash_R x_n$$

and, in the logical system R, we have a rule for deriving y using $x_1, \ldots, x_n$ as premises, i.e.,

$$\left(\frac{x_1, \ldots, x_n}{y}\right) \in R$$

then we can build a derivation for y as follows:

$$\left(\frac{\{d_1, \ldots, d_n\}}{y}\right) \Vdash_R y$$

Summarising all the above

- $(\varnothing / y) \Vdash_R y$ if $(\varnothing / y) \in R$ (axiom)
- $(\{d_1, \ldots, d_n\} / y) \Vdash_R y$ if $(\{x_1, \ldots, x_n\} / y) \in R$ and $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ (inference)

A derivation can roughly be seen as a tree whose root is the formula y we derive and whose leaves are the axioms we need. Correspondingly, we can define the height of a derivation tree as follows:

$$height(d) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } d = (\varnothing / y) \\ 1 + \max\{height(d_1), \ldots, height(d_n)\} & \text{if } d = (\{d_1, \ldots, d_n\} / y) \end{cases}$$

Definition 2.5 (Theorem). A *theorem* in a logical system R is a *well-formed formula* y for which there exists a proof, and we write $\Vdash_R y$.

In other words, y is a theorem in R if $\exists d. d \Vdash_R y$.

Definition 2.6 (Set of theorems in R). We let $I_R = \{y \mid \Vdash_R y\}$ be the set of all theorems that can be proved in R.

We mention two main approaches to prove theorems:

- *top-down* or *direct*: we start from theorems descending from the axioms and then prove more and more theorems by applying the inference rules to already proved theorems;
- *bottom-up* or *goal-oriented*: we fix a goal, i.e., a theorem we want to prove, and we try to deduce a derivation for it by applying the inference rules backwards, until each needed premise is also proved.

In the following we will mostly follow the *bottom-up* approach, because we will be given a specific goal to prove.

Example 2.5 (Grammars as sets of inference rules). Every grammar can be presented equivalently as a set of inference rules. Let us consider the well-known grammar for strings of balanced parentheses. Recalling that ε denotes the empty string, we write

$$S \quad ::= \quad S\,S \quad | \quad (S) \quad | \quad \varepsilon$$

We let L_S denote the set of strings generated by the grammar for the symbol S. The translation from production to inference rules is straightforward. The first production

$$S \quad ::= \quad S\,S$$

says that given any two strings s_1 and s_2 of balanced parentheses, their juxtaposition is also a string of balanced parentheses. In other words

$$\frac{s_1 \in L_S \quad s_2 \in L_S}{s_1 s_2 \in L_S}\,(1)$$

Similarly, the second production

$$S \quad ::= \quad (S)$$

says that we can surround with brackets any string s of balanced parentheses and get again a string of balanced parentheses. In other words:

$$\frac{s \in L_S}{(s) \in L_S}\,(2)$$

Finally, the last production says that the empty string ε is just a particular string of balanced parentheses. In other words we have an axiom:

$$\frac{}{\varepsilon \in L_S}\,(3)$$

Note the difference between the placeholders s, s_1, s_2 and the symbol ε appearing in the rules above: the former can be replaced by any string to obtain a specific instance of rules (1) and (2), while the latter denotes a given string (i.e., rules (1) and (2) define rule schemes with many instances, while there is a unique instance of rule (3)).

For example, the rule

$$\frac{)\,(\in L_S \quad (\,(\in L_S}{)\,(\,(\,(\in L_S}\,(1)$$

is an instance of rule (1): it is obtained by replacing s_1 with $)\,(\,$ and s_2 with $(\,(\,$. Of course the string $)\,(\,(\,(\,$ appearing in the conclusion does not belong to L_S, but the rule instance is perfectly valid, because it says that "$)\,(\,(\,(\in L_S$ *if* $)\,(\in L_S$ and $(\,(\in L_S$": since the premises are false, the implication is valid even if we cannot draw the conclusion $)\,(\,(\,(\in L_S$.

Let us see an example of valid derivation that uses some valid instances of rules (1) and (2):

$$\dfrac{\dfrac{\dfrac{}{\varepsilon \in L_S}\,(3)}{(\,\varepsilon\,) = (\,) \in L_S}\,(2)}{(\,(\,)\,) \in L_S}\,(2) \qquad \dfrac{\dfrac{}{\varepsilon \in L_S}\,(3)}{(\,\varepsilon\,) = (\,) \in L_S}\,(2)$$
$$\dfrac{\qquad\qquad}{(\,(\,)\,)\,(\,) \in L_S}\,(1)$$

Reading the proof (from the leaves to the root): since $\varepsilon \in L_S$ by axiom (3), then we know that $(\,\varepsilon\,) = (\,) \in L_S$ by (2); if we apply again rule (2) we derive also $(\,(\,)\,) \in L_S$ and hence $(\,(\,)\,)\,(\,) \in L_S$ by (1). In other words $(\,(\,)\,)\,(\,) \in L_S$ is a theorem.

Let us introduce a second formalisation of the same language (balanced parentheses) without using inference rules. To get some intuition, suppose we want to write an algorithm to check whether the parentheses in a string are balanced. We can parse the string from left to right and count the number of unmatched, open parentheses in the prefix we have parsed. So, we add 1 to the counter whenever we find an open parenthesis and subtract 1 whenever we find a closed parenthesis. If the counter is never negative, and it holds 0 when we have parsed the whole string, then the parentheses in the string are balanced.

In the following we let a_i denote the ith symbol of the string a. Let

$$f(a_i) = \begin{cases} 1 & \text{if } a_i = (\\ -1 & \text{if } a_i =) \end{cases}$$

A string of n parentheses $a = a_1 a_2 \ldots a_n$ is balanced if and only if both the following properties hold:

Property 1: $\forall m \in [0,n]$ we have $\sum_{i=1}^{m} f(a_i) \geq 0$
Property 2: $\sum_{i=1}^{n} f(a_i) = 0$

In fact, $\sum_{i=1}^{m} f(a_i)$ counts the difference between the number of open parentheses and closed parentheses that are present in the first m symbols of the string a. Therefore, the first property requires that in any prefix of the string a the number of open parentheses exceeds or equals the number of closed ones; the second property requires that the string a has as many open parentheses as closed ones.

An example is shown below for the string $a = (\,(\,)\,)\,(\,)$:

$$\begin{aligned} m &= 1\ 2\ 3\ 4\ 5\ 6 \\ a_m &= (\ (\ )\ )\ (\ ) \\ f(a_m) &= 1\ 1\ -1\ -1\ 1\ -1 \\ \textstyle\sum_{i=1}^{m} f(a_i) &= 1\ 2\ 1\ 0\ 1\ 0 \end{aligned}$$

Properties 1 and 2 are easy to check for any string and therefore define a useful procedure to decide whether a string belongs to our language or not.

Next, we show that the two different characterisations of the language (by inference rules and by the counting procedure) of balanced parentheses are equivalent.

Theorem 2.1. *For any string of parentheses a of length n*

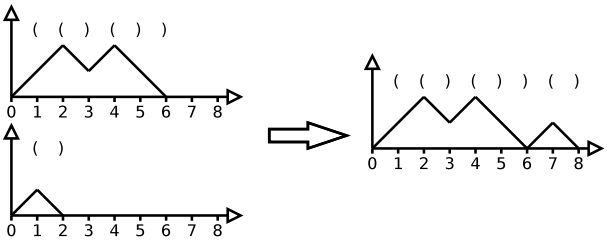
$$a \in L_S \iff \begin{cases} \sum_{i=1}^{m} f(a_i) \geq 0 & m = 0, 1 \ldots n \\ \sum_{i=1}^{n} f(a_i) = 0 \end{cases}$$

Proof. The proof is composed of two implications that we show separately:

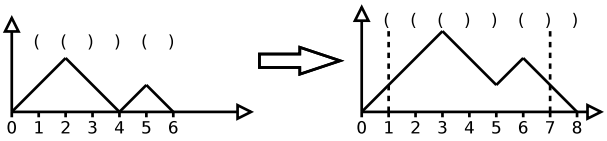
⇒) all the strings produced by the grammar satisfy the two properties;
⇐) any string that satisfies the two properties can be generated by the grammar.

Proof of ⇒) To show the first implication, we proceed by *induction over the rules*: we assume that the implication is valid for the premises and we show that it holds for the conclusion. This proof technique is very powerful and will be explained in detail in Chapter 4.
The two properties can be represented graphically over the cartesian plane by taking m over the x-axis and the quantity $\sum_{i=1}^{m} f(a_i)$ over the y-axis. Intuitively, the graph starts at the origin; it should never cross below the x-axis and it should end at $(n, 0)$.
Let us check that by applying any inference rule the properties 1 and 2 still hold.

Rule (1): The first inference rule corresponds to the juxtaposition of the two graphs and therefore the result still satisfies both properties (when the original graphs do).



Rule (2): The second rule corresponds to translating the graph upward (by one unit) and therefore the result still satisfies both properties (when the original graph does).

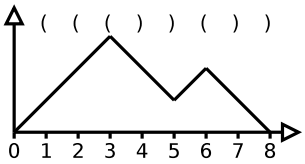


Rule (3): The third rule is just concerned with the empty string that trivially satisfies the two properties.

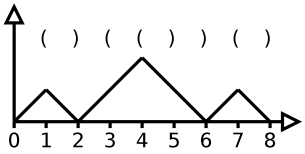
Since we have inspected all the inference rules, the proof of the first implication is concluded.

Proof of $\Leftarrow$) We need to find a derivation for any string that satisfies the two properties. Let a be such a generic string. (We only sketch this direction of the proof, which goes by induction over the length of the string a.) We proceed by case analysis, considering three cases:

1. If $n = 0$, $a = \varepsilon$. Then, by rule (3) we conclude that $a \in L_S$.
2. The second case is when the graph associated with a never touches the x-axis (except for its start and end points). An example is shown below:



   In this case we can apply rule (2), because we know that the parenthesis opened at the beginning of a is only matched by the parenthesis at the very end of a.
3. The third and last case is when the graph touches the x-axis (at least) once at a point $(k,0)$ different from its start and its end. An example is shown below:



   In this case the substrings $a_1 \ldots a_k$ and $a_{k+1} \ldots a_n$ are also balanced and we can apply the rule (1) to their derivations to prove that $a \in L_S$. □

The last part of the proof outlines a goal-oriented strategy to build a derivation for a given string: we start by looking for a rule whose conclusion can match the goal we are after. If there are no alternatives, then we fail. If we have only one alternative we need to build a derivation for its premises. If there are more alternatives than one we can either explore all of them in parallel (*breadth-first* approach) or try one of them and backtrack in case we fail (*depth-first*).

Suppose we want to find a proof for $(\,(\,)\,)\,(\,) \in L_S$. We use the notation

$$(\,(\,)\,)\,(\,) \in L_S \quad \nwarrow$$

to mean that we look for a goal-oriented derivation.

1. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; \varepsilon \in L_S,\quad (\,(\,)\,)\,(\,) \in L_S$
2. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; (\,\in L_S,\quad (\,)\,)\,(\,) \in L_S$
3. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; (\,(\,\in L_S,\quad)\,)\,(\,) \in L_S$
4. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; (\,(\,) \in L_S,\quad)\,(\,) \in L_S$
5. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; (\,(\,)\,) \in L_S,\quad (\,) \in L_S$
6. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; (\,(\,)\,)\,(\,\in L_S,\quad) \in L_S$
7. $(\,(\,)\,)\,(\,) \in L_S \;\nwarrow\; (\,(\,)\,)\,(\,) \in L_S,\quad \varepsilon \in L_S$

Fig. 2.1: Tentative derivations for the goal $(\,(\,)\,)\,(\,) \in L_S$

- Rule (1) can be applied in many different ways, by splitting the string $(\,(\,)\,)\,(\,)$ in all possible ways. We use the notation

$$(\,(\,)\,)\,(\,) \in L_S \quad\nwarrow\quad \varepsilon \in L_S,\ (\,(\,)\,)\,(\,) \in L_S$$

  to mean that we reduce the proof of $(\,(\,)\,)\,(\,) \in L_S$ to those of $\varepsilon \in L_S$ and $(\,(\,)\,)\,(\,) \in L_S$. Then we have all the alternatives in Figure 2.1 to inspect. Note that some alternatives are identical except for the order in which they list sub-goals (1 and 7) and may require us to prove the same goal from which we started (1 and 7). For example, if option 1 is selected applying depth-first strategy without any additional check, the derivation procedure might diverge. Moreover, some alternatives lead to goals we won't be able to prove (2, 3, 4, 6).
- Rule (2) can be applied in only one way:

$$(\,(\,)\,)\,(\,) \in L_S \quad\nwarrow\quad (\,)\,)\,(\,\in L_S$$

- Rule (3) cannot be applied.

We show below a successful derivation, where the empty goal is written $\square$:

$(\,(\,)\,)\,(\,) \in L_S$	$\nwarrow$	$(\,(\,)\,) \in L_S,\quad (\,) \in L_S$	by applying (1)
	$\nwarrow$	$(\,(\,)\,) \in L_S,\quad \varepsilon \in L_S$	by applying (2) to the second goal
	$\nwarrow$	$(\,(\,)\,) \in L_S$	by applying (3) to the second goal
	$\nwarrow$	$(\,) \in L_S$	by applying (2)
	$\nwarrow$	$\varepsilon \in L_S$	by applying (2)
	$\nwarrow$	$\square$	by applying (3)

We remark that in general the problem of checking whether a certain formula is a theorem is only *semi-decidable* (not necessarily *decidable*). In this case the breadth-first strategy for goal-oriented derivation offers a semi-decision procedure: if a derivation exists, then it will be found; if no derivation exists, the strategy may not terminate.

2.3 Logic Programming

We end this chapter by mentioning a particularly relevant paradigm based on goal-oriented derivation: *logic programming* and its Prolog incarnation. Prolog exploits depth-first goal-oriented derivations with backtracking.

Let $X = \{x, y, \ldots\}$ be a set of variables, $\Sigma = \{f, g, \ldots\}$ a signature of function symbols (with given arities), $\Pi = \{\mathsf{p}, \mathsf{q}, \ldots\}$ a signature of predicate symbols (with given arities). As usual, we denote by Σ_n (respectively Π_n) the subset of function symbols (respectively predicate symbols) with arity n.

Definition 2.7 (Atomic formula). An *atomic formula* consists of a predicate symbol p of arity n applied to n terms with variables.

For example, if $\mathsf{p} \in \Pi_2$, $f \in \Sigma_2$ and $g \in \Sigma_1$, then $\mathsf{p}(\, f(g(x), x)\,,\, g(y)\,)$ is an atomic formula.

Definition 2.8 (Formula). A *formula* is a (possibly empty) conjunction of atomic formulas.

Definition 2.9 (Horn clause). A *Horn clause* is written $l \colon\!- r$ where l is an atomic formula, called the *head* of the clause, and r is a formula called the *body* of the clause.

Definition 2.10 (Logic program). A logic program is a set of Horn clauses.

The variables appearing in each clause can be instantiated with any term. A goal g is a formula whose validity we want to prove. The goal g can contain variables, which are implicitly existentially quantified.

Unification is used to "match" the head of a clause to an atomic formula of the goal we want to prove in the most general way (i.e., by instantiating the variables as little as possible). Before performing unification, the variables of the clause are renamed with fresh identifiers to avoid any clash with the variables already present in the goal.

Suppose we are given a logic program L and a goal $g = a_1, \ldots, a_n$, where $a_1, \ldots, a_n$ are atomic formulas. A derivation step $g \nwarrow_{\sigma'} g'$ is obtained by selecting a sub-goal a_i, a clause $l \colon\!- r \in L$ and a renaming ρ such that:

- $l\rho \colon\!- r\rho$ is a variant of the clause $l \colon\!- r \in L$ whose variables are fresh;
- the unification problem $\{a_i \stackrel{?}{=} l\rho\}$ has a most general solution σ;
- $g' \stackrel{\text{def}}{=} a_1\sigma, \ldots, a_{i-1}\sigma, r\rho\sigma, a_{i+1}\sigma, \ldots, a_n\sigma$;
- $\sigma' \stackrel{\text{def}}{=} \sigma_{|vars(a_i)}$.

If we can find a sequence of derivation steps

$$g \nwarrow_{\sigma_1} g_1 \nwarrow_{\sigma_2} g_2 \cdots g_{n-1} \nwarrow_{\sigma_n} \square$$

then we can conclude that the goal g is satisfiable and that the substitution $\sigma \stackrel{\text{def}}{=} \sigma_1 \cdots \sigma_n$ is a least substitution for the variables in g such that $g\sigma$ is a valid theorem.

Example 2.6 (Sum in Prolog). Let us consider the logic program:

$$\begin{aligned} \mathsf{sum}(0, y, y) &: - \ . \\ \mathsf{sum}(s(x), y, s(z)) &: - \ \mathsf{sum}(x, y, z). \end{aligned}$$

where $\mathsf{sum} \in \Pi_3$, $s \in \Sigma_1$, $0 \in \Sigma_0$ and $x, y, z \in X$.

Let us consider the goal $\mathsf{sum}(s(s(0)), s(s(0)), v)$ with $v \in X$.

There is no match against the head of the first clause, because 0 does not unify with $s(s(0))$.

We rename x, y, z in the second clause to x', y', z' and compute the unification of $\mathsf{sum}(s(s(0)), s(s(0)), v)$ and $\mathsf{sum}(s(x'), y', s(z'))$. The result is the substitution

$$[x' = s(0), \quad y' = s(s(0)), \quad v = s(z')]$$

We then apply the substitution to the body of the clause, which will be added to the goal:

$$\mathsf{sum}(x', y', z')[x' = s(0), y' = s(s(0)), v = s(z')] = \mathsf{sum}(s(0), s(s(0)), z')$$

If other sub-goals were initially present, which may share variables with the goal $\mathsf{sum}(s(s(0)), s(s(0)), v)$, then the substitution should have been applied to them too.

We write the derivation described above using the notation

$$\mathsf{sum}(s(s(0)), s(s(0)), v) \quad \nwarrow_{v=s(z')} \quad \mathsf{sum}(s(0), s(s(0)), z')$$

where we have recorded (as a subscript of the derivation step) the substitution applied to the variables originally present in the goal (just v in the example), to record the least condition under which the derivation is possible.

The derivation can then be completed as follows:

$$\begin{array}{lll} \mathsf{sum}(s(s(0)), s(s(0)), v) & \nwarrow_{v=s(z')} & \mathsf{sum}(s(0), s(s(0)), z') \\ & \nwarrow_{z'=s(z'')} & \mathsf{sum}(0, s(s(0)), z'') \\ & \nwarrow_{z''=s(s(0))} & \square \end{array}$$

By composing the computed substitutions we get

$$\begin{aligned} z' &= s(z'') = s(s(s(0))) \\ v &= s(z') = s(s(s(s(0)))) \end{aligned}$$

This gives us a proof of the theorem

$$\mathsf{sum}(s(s(0)), s(s(0)), s(s(s(s(0)))))$$

Problems

2.1. Consider the alphabet $\{\mathtt{a},\mathtt{b}\}$ and the grammar

$$\begin{array}{lcl} A & ::= & \mathtt{a}\,A \mid \mathtt{a}\,B \\ B & ::= & \mathtt{b} \mid \mathtt{b}\,B \end{array}$$

1. Describe the form of the strings in the languages L_A and L_B.
2. Define the languages L_A and L_B formally.
3. Write the inference rules that correspond to the productions of the grammar.
4. Write the derivation for the string `a a a b b` both as a proof tree and as a goal-oriented derivation.
5. Prove that the set of theorems associated with the inference rules coincide with the formal definitions you gave.

2.2. Consider the alphabet $\{\mathtt{0},\mathtt{1}\}$.

1. Give a context-free grammar for the set of strings that contain an even number of `0`s and `1`s.
2. Write the inference rules that correspond to the productions of the grammar.
3. Write the derivation for the string `0 1 1 0 0 0` both as a proof tree and as a goal-oriented derivation.
4. Prove that your logical system characterises exactly the set of strings that contain an even number of `0`s and `1`s.

2.3. Consider the signature Σ such that $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ and $\Sigma_n = \varnothing$ for any $n \geq 2$.

1. Let $even \in \Pi_1$. What are the theorems of the logical system below?

$$\frac{}{even(0)}\,(1) \qquad \frac{even(x)}{even(s(s(x)))}\,(2)$$

2. Let $odd \in \Pi_1$.What are the theorems of the logical system below?

$$\frac{odd(x)}{odd(s(s(x)))}\,(1)$$

3. Let $leq \in \Pi_2$. What are the theorems of the logical system below?

$$\frac{}{leq(0,x)}\,(1) \qquad \frac{leq(x,y)}{leq(s(x),s(y))}\,(2)$$

2.4. Consider the signature Σ such that $\Sigma_0 = \mathbb{N}$, $\Sigma_2 = \{node\}$ and $\Sigma_n = \varnothing$ otherwise. Let $sum, eq \in \Pi_2$. What are the theorems of the logical system below?

$$\frac{}{sum(n,n)}\,n \in \mathbb{N}\,(1) \qquad \frac{sum(x,n) \quad sum(y,m)}{sum(node(x,y),k)}\,k = n+m\,(2) \qquad \frac{sum(x,n) \quad sum(y,n)}{eq(x,y)}\,(3)$$

2.5. Consider the signature Σ such that $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ and $\Sigma_n = \varnothing$ for any $n \geq 2$. Give two terms t and t', with $t \neq t'$, such that t is more general than t' and t' is also more general than t.

2.6. Consider the signature Σ such that $\Sigma_0 = \{a\}$, $\Sigma_1 = \{f,g\}$, $\Sigma_2 = \{h,l\}$ and $\Sigma_n = \varnothing$ for any $n \geq 3$. Solve the unification problems below:

1. $G_0 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(y,z), g(y) \stackrel{?}{=} g(l(a,a))\}$
2. $G_1 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(x,g(z))\}$
3. $G_2 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(y,f(z)), l(y,a) \stackrel{?}{=} l(a,z)\}$
4. $G_3 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(y,x) \stackrel{?}{=} h(g(a),f(g(z))), l(z,a) \stackrel{?}{=} l(a,z)\}$

2.7. Extend the logic program for computing the sum with the definition of

1. a predicate *prod* for computing the product of two numbers;
2. a predicate *pow* for computing the power of a base to an exponent;
3. a predicate *div* that tells whether a number can be divided by another number.

2.8. Extend the logic program for computing the sum with the definition of a binary predicate $\mathsf{fib}(N,F)$ that is true if F is the Nth Fibonacci number.

2.9. Suppose that a set of facts of the form $\mathsf{parent}(\mathsf{x},\mathsf{y})$ are given, meaning that x is a parent of y:

1. Define a predicate $\mathsf{sibling}(X,Y)$ which holds true iff X and Y have a parent in common.
2. Define a predicate $\mathsf{cousin}(X,Y)$ which holds true iff X and Y are cousins.
3. Define a predicate $\mathsf{ancestor}(X,Y)$ which holds true iff X is an ancestor of Y.
4. If the set of basic facts is:

```
parent(alice,bob) .
parent(alice,carl) .
parent(bob,diana) .
parent(bob,ella) .
parent(carl,francisco) .
```

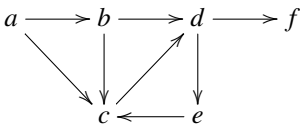
which of the following goals can be derived?

```
?- sibling(ella,francisco).
?- sibling(ella,diana).
?- cousin(ella,francisco).
?- cousin(ella,diana).
?- ancestor(alice,ella).
?- ancestor(carl,ella).
```

2.10. Suppose that a set of facts of the form $\mathsf{arc}(\mathsf{x},\mathsf{y})$ are given to represent a directed, acyclic graph, meaning that there is an arc from x to y:

1. Define a predicate $\mathsf{path}(X,Y)$ which holds true iff there is a path from X to Y.

2. Suppose the acyclic requirement is violated, as in the graph



defined by

```
arc(a,b) .
arc(a,c) .
arc(b,c) .
arc(b,d) .
arc(c,d) .
arc(d,e) .
arc(d,f) .
arc(e,c) .
```

Does a goal-oriented derivation for a query, such as the one below, necessarily lead to the empty goal? Why?

```
?- path(a,f).
```

2.11. Consider the Horn clauses that correspond to the following statements:

1. All jumping creatures are green.
2. All small jumping creatures are Martians.
3. All green Martians are intelligent.
4. Ngtrks is small and green.
5. Pgvdrk is a jumping Martian.

Who is intelligent?[1]

[1] Taken from `http://www.slideshare.net/SergeiWinitzki/prolog-talk`.

Part II
IMP: a Simple Imperative Language

This part focuses on models for sequential computations that are associated with IMP, a simple imperative language. The syntax and natural semantics of IMP are studied in Chapter 3, while its denotational semantics is presented in Chapter 6, where it is also reconciled with the operational semantics. Chapter 4 explains several induction principles exploited to prove properties of programs and semantics. Chapter 5 fixes the mathematical basis of denotational semantics. The concepts in Chapters 4 and 5 are extensively used in Chapter 6 and in the rest of the monograph.

Chapter 3
Operational Semantics of IMP

Programs must be written for people to read, and only incidentally for machines to execute. (H. Abelson and G. Sussman)

Abstract This chapter introduces the formal syntax and operational semantics of a simple, structured imperative language called IMP, with static variable allocation and no sophisticated declaration constructs for data types, functions, classes, methods and the like. The operational semantics is defined in the natural style and it assumes an abstract machine with a very basic form of memory to associate integer values with variables. The operational semantics is used to derive a notion of program equivalence and several examples of (in)equivalence proofs are shown.

3.1 Syntax of IMP

The IMP programming language is a simple imperative language (e.g., it can be seen as a bare-bones version of the C language) with only three data types:

int: the set of integer numbers, ranged over by metavariables $m, n, \ldots$

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

bool: the set of boolean values, ranged over by metavariables $u, v, \ldots$

$$\mathbb{B} = \{\textbf{true}, \textbf{false}\}$$

locations: the (denumerable) set of memory locations (we consider programs that use a finite number of locations and we assume there are enough locations available for any program), ranged over by metavariables $x, y, \ldots$ We shall use the terms location, variable and identifier interchangeably.

$$\textbf{Loc} \quad \mathit{locations}$$

The grammar for IMP comprises three syntactic categories:

Aexp: Arithmetic expressions, ranged over by $a, a', \ldots$

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_3

Bexp: Boolean expressions, ranged over by $b, b', \ldots$
Com : Commands, ranged over by $c, c', \ldots$

Definition 3.1 (IMP syntax). The following productions define the syntax of IMP:

$$\begin{array}{rcl} a \in Aexp & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ b \in Bexp & ::= & v \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\ c \in Com & ::= & \textbf{skip} \mid x := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c \end{array}$$

where we recall that n is an integer number, v a boolean value and x a location.

IMP is a very simple imperative language and there are several constructs we deliberately omit. For example we omit other common conditional statements, such as *switch*, and other cyclic constructs, such as *repeat*. Moreover IMP commands impose a structured flow of control, i.e., IMP has no labels, no goto statements, no break statements, no continue statements. Other things which are missing and are difficult to model are those concerned with *modular programming*. In particular, we have no *procedures*, no *modules*, no *classes*, no *types*. Since IMP does not include variable declarations, ambients, procedures and blocks, memory allocation is essentially static and finite, except for the possibility of handling integers of any size. Of course, IMP has no *concurrent programming* construct.

3.1.1 Arithmetic Expressions

An arithmetic expression can be an integer number, or a location, a sum, a difference or a product. We notice that we do not have division, because it can be undefined (e.g., $7/0$) or give different values (e.g., $0/0$) so that its use would introduce unnecessary complexity.

3.1.2 Boolean Expressions

A boolean expression can be a logical value v, or the equality of an arithmetic expression with another, whether an arithmetic expression is less than or equal to another one, a negation, a logical conjunction or disjunction.

To keep the notation compact, in the following we will take the liberty of writing boolean expressions such as $x \neq 0$ in place of $\neg(x = 0)$ and $x > 0$ in place of $1 \leq x$ or $(0 \leq x) \wedge \neg(x = 0)$.

3.1.3 Commands

A command can be **skip**, i.e. a command which does nothing, or an assignment, where we have that an arithmetic expression is evaluated and the value is assigned to a location; we can also have the sequential execution of two commands (one after the other); an **if-then-else** with the obvious meaning: we evaluate a boolean expression b, if it is true we execute c_0 and if it is false we execute c_1. Finally we have a **while** statement, which is a command that keeps executing c until b becomes false.

3.1.4 Abstract Syntax

The notation above gives the so-called *abstract syntax*, in that it simply says how to build up new expressions and commands but it is ambiguous for parsing a string. It is the job of the *concrete syntax* to provide enough information through parentheses or orders of precedence between operation symbols for a string to parse uniquely. It is helpful to think of a term in the abstract syntax as a specific parse tree of the language.

Example 3.1 (Valid expressions).

$(\textbf{while}\ b\ \textbf{do}\ c_1)\ ;\ c_2$	is a valid command;
$\textbf{while}\ b\ \textbf{do}\ (c_1\ ;\ c_2)$	is a valid command;
$\textbf{while}\ b\ \textbf{do}\ c_1\ ;\ c_2$	is not a valid command, because it is ambiguous.

In the following we will assume that enough parentheses have been added to resolve any ambiguity in the syntax. Then, given any formula of the form $a \in Aexp$, $b \in Bexp$ or $c \in Com$, the process to check whether this formula is a "theorem" is deterministic (no backtracking is needed).

Example 3.2 (Validity check). Let us consider the formula:

$$\textbf{if}\ (x=0)\ \textbf{then}\ (\textbf{skip})\ \textbf{else}\ (x := (x-1)) \in Com$$

We can prove its validity by the following (deterministic) derivation, where we write $\nwarrow^*$ to mean that several derivation steps are grouped into one for brevity:

$$\begin{aligned}
\textbf{if}\,(x=0)\,\textbf{then}\,(\textbf{skip})\,\textbf{else}\,(x := (x-1)) \in Com \ &\nwarrow\ x = 0 \in Bexp,\ \textbf{skip} \in Com,\\
&\qquad x := (x-1) \in Com\\
&\nwarrow\ x \in Aexp,\ 0 \in Aexp,\ \textbf{skip} \in Com,\\
&\qquad x := (x-1) \in Com\\
&\nwarrow^*\ x - 1 \in Aexp\\
&\nwarrow\ x \in Aexp, 1 \in Aexp\\
&\nwarrow^*\ \square
\end{aligned}$$

3.2 Operational Semantics of IMP

3.2.1 Memory State

In order to define the evaluation of an expression or the execution of a command, we need to handle the state of the machine which is going to execute the IMP statements. Beside expressions to be evaluated and commands to be executed, we also need to record in the state some additional elements such as values and stores. To this aim, we introduce the notion of *memory*:

$$\sigma \in \Sigma = (\mathbf{Loc} \to \mathbb{Z})$$

A memory σ is an element of the set Σ which contains all the functions from locations to integer numbers. A particular σ is just a function from locations to integer numbers so it is a function which associates with each location x the value $\sigma(x)$ that x stores.

Since **Loc** is an infinite set, things can be complicated: handling functions from an infinite set is not a good idea for a model of computation. Although **Loc** is large enough to store all the values that are manipulated by expressions and commands, the functions we are interested in are functions which are almost everywhere 0, except for a finite subset of memory locations.

If, for instance, we want to represent a memory such that the location x contains the value 5 and the location y the value 10 and elsewhere is stored 0, we write:

$$\sigma = (5/x,\ 10/y)$$

In this way we can represent any interesting memory by a finite set of pairs.

We let $()$ denote the memory such that all locations are assigned the value 0.

Definition 3.2 (Memory update). Given a memory σ, we denote by $\sigma[^n/_x]$ the memory where the value of x is updated to n, i.e. such that

$$\sigma[^n/_x](y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{if } y \neq x \end{cases}$$

Note that $\sigma[^n/_x][^m/_x] = \sigma[^m/_x]$. In fact

$$\sigma[^n/_x][^m/_x](y) = \begin{cases} m & \text{if } y = x \\ \sigma[^n/_x](y) = \sigma(y) & \text{if } y \neq x \end{cases}$$

Moreover, when $x \neq y$, then the order of updates is not important, i.e., $\sigma[^n/_x][^m/_y] = \sigma[^m/_y][^n/_x]$. For this reason, we often use the more compact notation $\sigma[^n/_x, ^m/_y]$.

3.2.2 Inference Rules

Now we are going to give the *operational semantics* of IMP using a logical system. It is called "big-step" semantics (see Section 1.2) because it leads to the result in one single proof.

We are interested in three kinds of *well-formed formulas*:

Arithmetic expressions: The evaluation of an element $a \in Aexp$ in a given memory σ results in an integer number.

$$\langle a, \sigma \rangle \to n$$

Boolean expressions: The evaluation of an element $b \in Bexp$ in a given memory σ results in either **true** or **false**.

$$\langle b, \sigma \rangle \to v$$

Commands: The evaluation of an element $c \in Com$ in a given memory σ leads to an updated final state σ'.

$$\langle c, \sigma \rangle \to \sigma'$$

Next we show each inference rule and comment on it.

3.2.2.1 Inference Rules for Arithmetic Expressions

We start with the rules about arithmetic expressions.

$$\frac{}{\langle n, \sigma \rangle \to n}\,(\text{num}) \tag{3.1}$$

The axiom 3.1 (num) is trivial: the evaluation of any numerical constant n (seen as syntax) results in the corresponding integer value n (read as an element of the semantic domain) no matter which σ.

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)}\,(\text{ide}) \tag{3.2}$$

The axiom 3.2 (ide) is also quite intuitive: the evaluation of an identifier x in the memory σ results in the value stored in x.

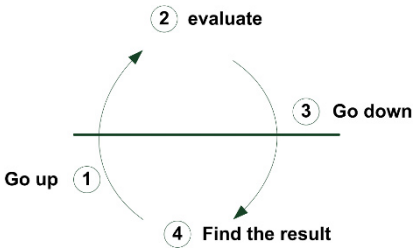
$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n}\, n = n_0 + n_1\,(\text{sum}) \tag{3.3}$$

The rule 3.3 (sum) has several premises: the evaluation of the syntactic expression $a_0 + a_1$ in σ returns a value n that corresponds to the arithmetic sum of the values n_0 and n_1 obtained after evaluating, respectively, a_0 and a_1 in σ. Note that we exploit the side condition $n = n_0 + n_1$ to indicate the relation between the target n of the conclusion and the targets of the premises. We present an equivalent, but more compact, version of the rule (sum), where the target of the conclusion is obtained as the sum of the targets of the premises. In the following we shall adopt the second format (3.4).

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n_0 + n_1}\ (\text{sum}) \tag{3.4}$$

We remark on the difference between the two occurrences of the symbol $+$ in the rule: in the source of the conclusion (i.e., $a_0 + a_1$) it denotes a piece of syntax, in the target of the conclusion (i.e., $n_0 + n_1$) it denotes a semantic operation. To avoid any ambiguity we could have introduced different symbols in the two cases, but we have preferred to overload the symbol and keep the notation simpler. We hope the reader is expert enough to assign the right meaning to each occurrence of overloaded symbols by looking at the context in which they appear.

The way we read this rule is very interesting because, in general, if we want to evaluate the lower part we have to go up, evaluate the upper part and then compose the results and finally go down again to draw the conclusion:



In this case we suppose we want to evaluate, in the memory σ, the arithmetic expression $a_0 + a_1$. We have to evaluate a_0 in the same memory σ and get n_0, then we have to evaluate a_1 within the same memory σ to get n_1 and then the final result will be $n_0 + n_1$. Note that the same memory σ is duplicated and distributed to the two evaluations of a_0 and a_1, which may occur independently in any order.

This kind of mechanism is very powerful because we deal with more proofs at once. First, we evaluate a_0. Second, we evaluate a_1. Then, we put it all together. If we need to evaluate several expressions on a sequential machine we have to deal with the issue of fixing the order in which to proceed. On the other hand, in this case, using a logical language we just model the fact that we want to evaluate a tree (an expression) which is a tree of proofs in a very simple way and make explicit that the order is not important.

The rules for the remaining arithmetic expressions are similar to the one for sum. We report them for completeness, but do not comment on them:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 - a_1, \sigma\rangle \to n_0 - n_1}\ (\text{dif}) \tag{3.5}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}\ (\text{prod}) \tag{3.6}$$

3.2.2.2 Inference Rules for Boolean Expressions

The rules for boolean expressions are also similar to the previous ones and need no particular comment, except for noting that the premises of rules (equ) and (leq) refer to judgements of arithmetic expressions:

$$\frac{}{\langle v, \sigma\rangle \to v}\ (\text{bool}) \tag{3.7}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to (n_0 = n_1)}\ (\text{equ}) \tag{3.8}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to (n_0 \leq n_1)}\ (\text{leq}) \tag{3.9}$$

$$\frac{\langle b, \sigma\rangle \to v}{\langle \neg b, \sigma\rangle \to \neg v}\ (\text{not}) \tag{3.10}$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \vee b_1, \sigma\rangle \to (v_0 \vee v_1)}\ (\text{or}) \tag{3.11}$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \wedge b_1, \sigma\rangle \to (v_0 \wedge v_1)}\ (\text{and}) \tag{3.12}$$

3.2.2.3 Inference Rules for Commands

Next, we move to the inference rules for commands.

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma} \text{(skip)} \tag{3.13}$$

The rule 3.13 (skip) is very simple: it leaves the memory σ unchanged.

$$\frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma[{}^{m}/_{x}]} \text{(assign)} \tag{3.14}$$

The rule 3.14 (assign) exploits the assignment operation to update σ: we recall that $\sigma[{}^{m}/_{x}]$ is the same memory as σ except for the value assigned to x (m instead of $\sigma(x)$). Note that the premise refers to the judgement of an arithmetic expression:

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'} \text{(seq)} \tag{3.15}$$

The rule 3.15 (seq) for the sequential composition (concatenation) of commands is quite interesting. We start by evaluating the first command c_0 in the memory σ. As a result we get an updated memory σ'' which we use for evaluating the second command c_1. In fact the order of evaluation of the two commands is important and it would not make sense to evaluate c_1 in the original memory σ, because the effects of executing c_0 would be lost. Finally, the memory σ' obtained by evaluating c_1 in σ'' is returned as the result of evaluating $c_0; c_1$ in σ.

The conditional statement requires two different rules; which is used depends on the evaluation of the condition b (they are mutually exclusive):

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'} \text{(iftt)} \tag{3.16}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'} \text{(ifff)} \tag{3.17}$$

The rule 3.16 (iftt) checks that b evaluates to true and then returns as result the memory σ' obtained by evaluating the command c_0 in σ. On the contrary, the rule 3.17 (ifff) checks that b evaluates to false and then returns as result the memory σ' obtained by evaluating the command c_1 in σ.

Also the while statement requires two different rules; which is used depends on the evaluation of the guard b; they are mutually exclusive:

$$\frac{\langle b,\sigma\rangle \to \textbf{true} \quad \langle c,\sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c,\sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \to \sigma'}\ (\text{whtt}) \tag{3.18}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \to \sigma}\ (\text{whff}) \tag{3.19}$$

The rule 3.18 (whtt) applies to the case where the guard evaluates to true: we need to compute the memory σ'' obtained by the evaluation of the body c in σ and then to iterate the evaluation of the cycle over σ''.

The rule 3.19 (whff) applies to the case where the guard evaluates to false: then the cycle terminates and the memory σ is returned unchanged.

Remark 3.1. There is an important difference between the rule 3.18 and all the other inference rules we have encountered so far. All the other rules take as premises formulas that are "smaller in size" than their conclusions. This fact allows us to decrease the complexity of the atomic goals to be proved as the derivation proceeds further, until we have basic formulas to which axioms can be applied. The rule 3.18 is different because it recursively uses as a premise a formula as complex as its conclusion. This justifies the fact that a while command can cycle indefinitely, without terminating.

The set of all inference rules above defines the operational semantics of IMP. Formally, they induce a relation that contains all the pairs input-result, where the input is the expression/command together with the initial memory and the result is the corresponding evaluation:

$$\to \subseteq (Aexp \times \Sigma \times \mathbb{Z}) \cup (Bexp \times \Sigma \times \mathbb{B}) \cup (Com \times \Sigma \times \Sigma)$$

We will see later that the computation is deterministic, in the sense that given any expression/command and any memory as input there is at most one result (exactly one in the case of arithmetic and boolean expressions).

3.2.3 Examples

Example 3.3 (Semantic evaluation of a command). Let us consider the (extra-bracketed) command

$$c \stackrel{\text{def}}{=} (x := 0)\ ;\ (\ \textbf{while}\ (0 \le y)\ \textbf{do}\ (\ (x := ((x + (2 \times y)) + 1))\ ;\ (y := (y - 1))\ )\ )$$

To improve readability and without introducing too much ambiguity, we can write it as follows:

$$c \stackrel{\text{def}}{=} x := 0 \; ; \; \textbf{while } 0 \le y \textbf{ do } (\; x := x + (2 \times y) + 1 \; ; \; y := y - 1 \;)$$

or exploiting the usual convention for indented code, as

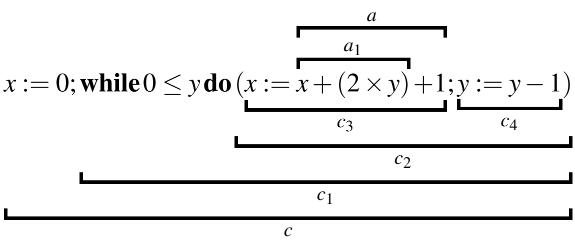
$$
\begin{array}{l}
c \stackrel{\text{def}}{=} x := 0 \; ; \\
\quad \textbf{while } 0 \le y \textbf{ do } (\\
\qquad x := x + (2 \times y) + 1 \; ; \\
\qquad y := y - 1 \\
\quad)
\end{array}
$$

Without too much difficulty, the experienced reader can guess the relation between the value of y at the beginning of the execution and that of x at the end of the execution: the program computes the square of (the value initially stored in) y plus 1 (when $y \ge 0$) and stores it in x. In fact, by exploiting the well-known equalities $0^2 = 0$ and $(n+1)^2 = n^2 + 2n + 1$, the value of $(y+1)^2$ is computed as the sum of the first $y+1$ odd numbers $\sum_{i=0}^{y}(2i+1)$. For example, for $y = 3$ we have $4^2 = 1+3+5+7 = 16$.

We report below the proof of well-formedness of the command, as a witness that c respects the syntax of IMP. (Of course the inference rules used in the derivation are those associated with the productions of the grammar of IMP.)

$$
\dfrac{\dfrac{\dfrac{}{x}\quad\dfrac{}{0}}{x := 0}\qquad \dfrac{\dfrac{\dfrac{}{0}\quad\dfrac{}{y}}{0 \le y}\qquad \dfrac{\dfrac{\dfrac{}{x}\quad \dfrac{\dfrac{\dfrac{}{x}\quad\dfrac{\dfrac{}{2}\quad\dfrac{}{y}}{2\times y}}{a_1 \stackrel{\text{def}}{=} (x+(2\times y))}\quad \dfrac{}{1}}{a \stackrel{\text{def}}{=} ((x+(2\times y))+1)}}{c_3 \stackrel{\text{def}}{=} (x := ((x+(2\times y))+1))}\qquad \dfrac{\dfrac{}{y}\quad\dfrac{\dfrac{}{y}\quad\dfrac{}{1}}{y-1}}{c_4 \stackrel{\text{def}}{=} (y := (y-1))}}{c_2 \stackrel{\text{def}}{=} ((x := ((x+(2\times y))+1)); (y := (y-1)))}}{c_1 \stackrel{\text{def}}{=} (\textbf{while}(0 \le y)\,\textbf{do}((x := ((x+(2\times y))+1)); (y := (y-1))))}}{c \stackrel{\text{def}}{=} ((x := 0); (\textbf{while}(0 \le y)\,\textbf{do}((x := ((x+(2\times y))+1)); (y := (y-1)))))}
$$

We can summarise the above proof as follows, introducing several shorthands to refer to some subterms of c that will be useful later:



To find the semantics of c in a given memory we proceed in a goal-oriented fashion. For instance, we take the well-formed formula $\langle c, \binom{27}{x}, \binom{2}{y} \rangle \to \sigma$, with σ unknown, and check whether there exists a memory σ such that the formula becomes a theorem. This is equivalent to finding an answer to the following question: "given the initial memory $\binom{27}{x}, \binom{2}{y}$ and the command c to be executed, can we find a derivation that leads to some memory σ?" By answering in the affirmative, we would have a proof of termination for c and would establish the content of the memory at the end of the computation.

To convince the reader that the notation for goal-oriented derivations introduced in Section 2.3 is more effective than the tree-like notation, we first show the proof in the tree-like notation: the goal to prove is the root (situated at the bottom) and the "pieces" of derivation are added on top. As the tree rapidly grows large, we split the derivation into smaller pieces that are proved separately. We use "?" to mark the missing parts of the derivations:

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \binom{27}{x}, \binom{2}{y} \rangle \to 0}\,\text{num}}{\langle x := 0, \binom{27}{x}, \binom{2}{y} \rangle \to \binom{27}{x}, \binom{2}{y}\left[\binom{0}{x}\right] = \sigma_1}\,\text{assign} \qquad \dfrac{?}{\langle c_1, \sigma_1 \rangle \to \sigma}}{\langle c, \binom{27}{x}, \binom{2}{y} \rangle \to \sigma}\,\text{seq}$$

Note that c_1 is a cycle, therefore we have two possible rules that can be applied, depending on the evaluation of the guard. We only show the successful derivation, recalling that $\sigma_1 = \binom{27}{x}, \binom{2}{y}\left[\binom{0}{x}\right] = \binom{0}{x}, \binom{2}{y}$:

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \sigma_1 \rangle \to 0}\,\text{num} \qquad \dfrac{}{\langle y, \sigma_1 \rangle \to \sigma_1(y) = 2}\,\text{ide}}{\langle 0 \le y, \sigma_1 \rangle \to (0 \le 2) = \textbf{true}}\,\text{leq} \qquad \dfrac{?}{\langle c_2, \sigma_1 \rangle \to \sigma_2} \qquad \dfrac{?}{\langle c_1, \sigma_2 \rangle \to \sigma}}{\langle c_1, \sigma_1 \rangle \to \sigma}\,\text{whtt}$$

Next we need to prove the goals $\langle c_2, \binom{0}{x}, \binom{2}{y} \rangle \to \sigma_2$ and $\langle c_1, \sigma_2 \rangle \to \sigma$. Let us focus on $\langle c_2, \sigma_1 \rangle \to \sigma_2$ first:

$$\dfrac{\dfrac{\dfrac{\dfrac{?}{\langle a_1, \binom{0}{x}, \binom{2}{y} \rangle \to m'} \qquad \dfrac{}{\langle 1, \binom{0}{x}, \binom{2}{y} \rangle \to 1}\,\text{num}}{\langle a, \binom{0}{x}, \binom{2}{y} \rangle \to m = m' + 1}\,\text{sum}}{\langle c_3, \binom{0}{x}, \binom{2}{y} \rangle \to \binom{0}{x}, \binom{2}{y}\left[\binom{m}{x}\right] = \sigma_3}\,\text{assign} \qquad \dfrac{\dfrac{?}{\langle y - 1, \sigma_3 \rangle \to m''}}{\langle c_4, \sigma_3 \rangle \to \sigma_3\left[\binom{m''}{y}\right] = \sigma_2}\,\text{assign}}{\langle c_2, \binom{0}{x}, \binom{2}{y} \rangle \to \sigma_2}\,\text{seq}$$

We show separately the details for the pending derivations of $\langle a_1, \binom{0}{x}, \binom{2}{y} \rangle \to m'$ and $\langle y - 1, \sigma_3 \rangle \to m''$:

$$\dfrac{\dfrac{}{\langle x, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 0}\,\text{ide} \qquad \dfrac{\dfrac{}{\langle 2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\,\text{num} \qquad \dfrac{}{\langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\,\text{ide}}{\langle 2 \times y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m''' = 2 \times 2 = 4}\,\text{prod}}{\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m' = 0 + 4 = 4}\,\text{sum}$$

Since $m' = 4$, this means that $m = m' + 1 = 5$ and hence $\sigma_3 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{5}/_{x}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)$.

$$\dfrac{\dfrac{}{\langle y, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\,\text{ide} \qquad \dfrac{}{\langle 1, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to 1}\,\text{num}}{\langle y - 1, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to m'' = 2 - 1 = 1}\,\text{dif}$$

Since $m'' = 1$ we know that $\sigma_2 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{m''}/_{y}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{1}/_{y}\right] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)$.

Next we prove $\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma$, this time omitting some details (the derivation is analogous to the one just seen):

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \textbf{true}}\,\text{leq} \qquad \dfrac{\vdots}{\langle c_2, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \left({}^{8}/_{x}, {}^{0}/_{y}\right) = \sigma_4}\,\text{seq} \qquad \dfrac{?}{\langle c_1, \sigma_4\rangle \to \sigma}}{\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma}\,\text{whtt}$$

Hence $\sigma_4 = \left({}^{8}/_{x}, {}^{0}/_{y}\right)$ and next we prove $\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma$:

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \textbf{true}}\,\text{leq} \qquad \dfrac{\vdots}{\langle c_2, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right) = \sigma_5}\,\text{seq} \qquad \dfrac{?}{\langle c_1, \sigma_5\rangle \to \sigma}}{\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma}\,\text{whtt}$$

Hence $\sigma_5 = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)$. Finally

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \textbf{false}}\,\text{leq}}{\langle c_1, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right) = \sigma}\,\text{whff}$$

Summing up all the above, we have proved the theorem

$$\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right).$$

It is evident that as the proof tree grows larger it gets harder to paste the different pieces of the proof together. We now show the same proof as a goal-oriented derivation, which should be easier to follow. To this aim, we group several derivation steps into a single one (written $\nwarrow^*$) omitting trivial steps:

$$
\begin{aligned}
\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma \quad &\nwarrow \quad \langle x := 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_1,\ \langle c_1, \sigma_1\rangle \to \sigma \\
&\nwarrow_{\sigma_1 = \left({}^{27}/_{x}, {}^{2}/_{y}\right)[{}^{n}/_{x}]} \quad \langle 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to n,\ \langle c_1, \left({}^{27}/_{x}, {}^{2}/_{y}\right)[{}^{n}/_{x}]\rangle \to \sigma \\
&\nwarrow_{n=0,\ \sigma_1 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)} \quad \langle c_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma \\
&\nwarrow \quad \langle 0 \le y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \mathbf{true}, \\
&\qquad\quad \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow \quad \langle 0, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to n_1,\ \langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to n_2, \\
&\qquad\quad n_1 \le n_2,\ \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow^*_{n_1=0,\ n_2=2} \quad \langle c_3, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_3,\ \langle c_4, \sigma_3\rangle \to \sigma_2, \\
&\qquad\quad \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow_{\sigma_3 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)[{}^{m}/_{x}]} \quad \langle x + (2 \times y) + 1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m, \\
&\qquad\quad \langle c_4, \left({}^{0}/_{x}, {}^{2}/_{y}\right)[{}^{m}/_{x}]\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow^*_{m=0+(2\times 2)+1=5,\ \sigma_3 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)} \quad \langle c_4, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow^*_{\sigma_2 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)[{}^{1}/_{y}] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)} \quad \langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma \\
&\nwarrow^*_{\sigma_4 = \left({}^{5}/_{x}, {}^{1}/_{y}\right)[{}^{8}/_{x}][{}^{0}/_{y}] = \left({}^{8}/_{x}, {}^{0}/_{y}\right)} \quad \langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma \\
&\nwarrow^*_{\sigma_5 = \left({}^{8}/_{x}, {}^{0}/_{y}\right)[{}^{9}/_{x}][{}^{-1}/_{y}] = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)} \quad \langle c_1, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \sigma \\
&\nwarrow_{\sigma = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)} \quad \langle 0 \le y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \mathbf{false} \\
&\nwarrow^* \quad \Box
\end{aligned}
$$

There are commands c and memories σ such that there is no σ' for which we can find a proof of $\langle c, \sigma\rangle \to \sigma'$. We use the notation below to denote such cases:

$$\langle c, \sigma\rangle \not\to \quad \text{iff} \quad \neg\exists \sigma'. \langle c, \sigma\rangle \to \sigma'$$

The condition $\neg\exists\sigma'.\langle c, \sigma\rangle \to \sigma'$ can be written equivalently as $\forall\sigma'.\langle c, \sigma\rangle \not\to \sigma'$.

Example 3.4 (Non-termination). Let us consider the command

$$c \stackrel{\text{def}}{=} \textbf{while true do skip}$$

Given σ, the only possible derivation goes as follows:

$$\begin{aligned}\langle c,\sigma\rangle \to \sigma' &\quad\nwarrow\ \langle \mathbf{true},\sigma\rangle \to \mathbf{true},\ \langle \mathbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\ &\quad\nwarrow\ \langle \mathbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\ &\nwarrow_{\sigma_1=\sigma}\ \langle c,\sigma\rangle \to \sigma'\end{aligned}$$

After a few steps of derivation we reach the same goal from which we started and there are no alternatives to try!

In this case, we can prove that $\langle c,\sigma\rangle \not\to$. We proceed by contradiction, assuming there exists σ' for which we can find a (finite) derivation d for $\langle c,\sigma\rangle \to \sigma'$. Let d be the derivation sketched below:

$$\begin{array}{l}\langle c,\sigma\rangle \to \sigma' \nwarrow \langle \mathbf{true},\sigma\rangle \to \mathbf{true},\ \langle \mathbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\ \qquad\vdots \\ (*) \nwarrow \langle c,\sigma\rangle \to \sigma' \\ \qquad\vdots \\ \nwarrow \Box\end{array}$$

We have marked by $(*)$ the last occurrence of the goal $\langle c,\sigma\rangle \to \sigma'$. But this leads to a contradiction, because the next step of the derivation can only be obtained by applying rule (whtt) and therefore it must lead to another instance of the original goal.

3.3 Abstract Semantics: Equivalence of Expressions and Commands

In the same way as we can write different expressions denoting the same value, we can write different programs for solving the same problem. For example we are used to not distinguishing between say $2+2$ and 2×2 because both evaluate to 4. Similarly, would you distinguish between, say, $x := 1; y := 0$ and $y := 0; x := y+1$? So a natural question arises: when are two programs "equivalent"? The equivalence between two commands is an important issue because it allows, e.g., to replace a program with an equivalent but more efficient one. Informally, two programs are equivalent if they *behave in the same way*. But can we make this idea more precise?

Since the evaluation of a command depends on the memory, two equivalent programs must behave the same *w.r.t. any initial memory*. For example the two commands $x := 1$ and $x := y+1$ assign the same value to x only when evaluated in a memory σ such that $\sigma(y) = 0$, so that it wouldn't be safe to replace one with the other in an arbitrary program. Moreover, we must take into account that commands may diverge when evaluated with a certain memory, such as **while** $x \neq 0$ **do** $x := x-1$ when evaluated in a store σ such that $\sigma(x) < 0$. We will call *abstract semantics* the notion of behaviour w.r.t. which we will compare programs for equivalence.

The operational semantics offers a straightforward abstract semantics: two programs are equivalent if they result in the same memory when evaluated over the same initial memory.

Definition 3.3 (Equivalence of expressions and commands). We say that the arithmetic expressions a_1 and a_2 are *equivalent*, written $a_1 \sim a_2$, if and only if for any memory σ they evaluate in the same way. Formally

$$a_1 \sim a_2 \quad \text{iff} \quad \forall \sigma, n.(\ \langle a_1, \sigma \rangle \to n \ \Leftrightarrow\ \langle a_2, \sigma \rangle \to n\ )$$

We say that the boolean expressions b_1 and b_2 are *equivalent*, written $b_1 \sim b_2$, if and only if for any memory σ they evaluate in the same way. Formally

$$b_1 \sim b_2 \quad \text{iff} \quad \forall \sigma, v.(\ \langle b_1, \sigma \rangle \to v \ \Leftrightarrow\ \langle b_2, \sigma \rangle \to v\ )$$

We say that the commands c_1 and c_2 are *equivalent*, written $c_1 \sim c_2$, if and only if for any memory σ they evaluate in the same way. Formally

$$c_1 \sim c_2 \quad \text{iff} \quad \forall \sigma, \sigma'.(\ \langle c_1, \sigma \rangle \to \sigma' \ \Leftrightarrow\ \langle c_2, \sigma \rangle \to \sigma'\ )$$

Note that if the evaluation of $\langle c_1, \sigma \rangle$ diverges there is no σ' such that $\langle c_1, \sigma \rangle \to \sigma'$. Then, when $c_1 \sim c_2$, the double implication prevents $\langle c_2, \sigma \rangle$ from converging. As an easy consequence, any two programs that diverge for all σ are equivalent.

3.3.1 Examples: Simple Equivalence Proofs

The first example we show is concerned with fully specified programs that operate on unspecified memories.

Example 3.5 (Equivalent commands). Let us try to prove that the following two commands are equivalent:

$$c_1 \stackrel{\text{def}}{=} \textbf{while}\ x \neq 0\ \textbf{do}\ x := 0$$
$$c_2 \stackrel{\text{def}}{=} x := 0$$

It is immediate to prove that

$$\forall \sigma. \langle c_2, \sigma \rangle \to \sigma' = \sigma[{}^{0}/_{x}]$$

Hence σ and σ' can differ only for the value stored in x. In particular, if $\sigma(x) = 0$ then $\sigma' = \sigma$.

The evaluation of c_1 on σ depends on $\sigma(x)$: if $\sigma(x) = 0$ we must apply the rule 3.19 (whff), otherwise the rule 3.18 (whtt) must be applied. Since we do not know the value of $\sigma(x)$, we consider the two cases separately. The corresponding hypotheses are called *path conditions* and outline a very important technique for the symbolic analysis of programs.

Case $\sigma(x) \neq 0$) Let us inspect a possible derivation for $\langle c_1, \sigma\rangle \to \sigma'$. Since $\sigma(x) \neq 0$ we select the rule (whtt) at the first step:

$$\begin{array}{rl} \langle c_1, \sigma\rangle \to \sigma' \quad \nwarrow & \langle x \neq 0, \sigma\rangle \to \mathbf{true}, \quad \langle x := 0, \sigma\rangle \to \sigma_1, \\ & \langle c_1, \sigma_1\rangle \to \sigma' \\ \nwarrow^{*}_{\sigma_1 = \sigma\left[{}^{0}/_{x}\right]} & \langle c_1, \sigma\left[{}^{0}/_{x}\right]\rangle \to \sigma' \\ \nwarrow_{\sigma' = \sigma\left[{}^{0}/_{x}\right]} & \langle x \neq 0, \sigma\left[{}^{0}/_{x}\right]\rangle \to \mathbf{false} \\ \nwarrow^{*} & \sigma\left[{}^{0}/_{x}\right](x) = 0 \\ \nwarrow & \square \end{array}$$

Case $\sigma(x) = 0$) Let us inspect a derivation for $\langle c_1, \sigma\rangle \to \sigma'$. Since $\sigma(x) = 0$ we select the rule (whff) at the first step:

$$\begin{array}{rl} \langle c_1, \sigma\rangle \to \sigma' \quad \nwarrow_{\sigma' = \sigma} & \langle x \neq 0, \sigma\rangle \to \mathbf{false} \\ \nwarrow^{*} & \sigma(x) = 0 \\ \nwarrow & \square \end{array}$$

Finally, we observe the following:

- If $\sigma(x) = 0$, then $\begin{cases} \langle c_1, \sigma\rangle \to \sigma \\ \langle c_2, \sigma\rangle \to \sigma\left[{}^{0}/_{x}\right] = \sigma \end{cases}$
- Otherwise, if $\sigma(x) \neq 0$, then $\begin{cases} \langle c_1, \sigma\rangle \to \sigma\left[{}^{0}/_{x}\right] \\ \langle c_2, \sigma\rangle \to \sigma\left[{}^{0}/_{x}\right] \end{cases}$

Therefore $c_1 \sim c_2$ because for any σ they result in the same memory.

The general methodology should be clear by now: in case the computation terminates we just need to develop the derivation and compare the results.

3.3.2 Examples: Parametric Equivalence Proofs

The programs considered so far were entirely spelled out: all the commands and expressions were given and the only unknown parameter was the initial memory σ. In this section we address equivalence proofs for programs that contain symbolic expressions a and b and symbolic commands c: we will need to prove that equality holds for any such a, b and c.

This is not necessarily more complicated than what we have done already: the idea is that we can just carry the derivation with symbolic parameters.

Example 3.6 (Parametric proofs (1)). Let us consider the commands

$c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$

$c_2 \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip} = \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}$

Is it true that $\forall b \in Bexp, c \in Com.\ (c_1 \sim c_2)$?

We start by considering the derivation for c_1 in a generic initial memory σ. The command c_1 is a cycle and there are two rules we can apply: either the rule 3.19 (whff), or the rule 3.18 (whtt). Which rule to use depends on the evaluation of b. Since we do not know what b is, we must take into account both possibilities and consider the two cases separately.

$\langle b, \sigma \rangle \to \textbf{false}$) For c_1 we have

$$\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma' \quad \nwarrow_{\sigma' = \sigma} \quad \langle b, \sigma \rangle \to \textbf{false}$$
$$\nwarrow \ \square$$

For c_2 we have

$$\langle \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}, \sigma \rangle \to \sigma' \quad \nwarrow \quad \langle b, \sigma \rangle \to \textbf{false}, \quad \langle \textbf{skip}, \sigma \rangle \to \sigma'$$
$$\nwarrow^{*}_{\sigma' = \sigma} \ \square$$

It is evident that if $\langle b, \sigma \rangle \to \textbf{false}$ then the two derivations for c_1 and c_2 lead to the same result.

$\langle b, \sigma \rangle \to \textbf{true}$) For c_1 we have

$$\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma' \ \nwarrow \ \langle b, \sigma \rangle \to \textbf{true}, \quad \langle c, \sigma \rangle \to \sigma_1, \quad \langle c_1, \sigma_1 \rangle \to \sigma'$$
$$\nwarrow \ \langle c, \sigma \rangle \to \sigma_1, \quad \langle c_1, \sigma_1 \rangle \to \sigma'$$

We find it convenient to stop the derivation here, because otherwise we should add further hypotheses on the evaluation of c and of the guard b after the execution of c. Instead, let us look at the derivation of c_2:

$$\langle \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}, \sigma \rangle \to \sigma' \ \nwarrow \ \langle b, \sigma \rangle \to \textbf{true}, \quad \langle c; c_1, \sigma \rangle \to \sigma'$$
$$\nwarrow \ \langle c; c_1, \sigma \rangle \to \sigma'$$
$$\nwarrow \ \langle c, \sigma \rangle \to \sigma_1, \quad \langle c_1, \sigma_1 \rangle \to \sigma'$$

Now we can stop again, because we have reached exactly the same sub-goals that we have obtained by evaluating c_1! It is then obvious that if $\langle b, \sigma \rangle \to \mathbf{true}$ then the two derivations for c_1 and c_2 will necessarily lead to the same result whenever they terminate, and if one diverges the other diverges too.

Summing up the two cases, and since there are no more alternatives to try, we can conclude that $c_1 \sim c_2$.

Note that the equivalence proof technique that exploits reduction to the same sub-goals is one of the most convenient methods for proving the equivalence of **while** commands, whose evaluation may diverge.

Example 3.7 (Parametric proofs (2)). Let us consider the commands

$$c_1 \stackrel{\text{def}}{=} \mathbf{while}\ b\ \mathbf{do}\ c$$
$$c_2 \stackrel{\text{def}}{=} \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else\ skip}$$

Is it true that $\forall b \in Bexp, c \in Com.\ c_1 \sim c_2$?

We have already examined the different derivations for c_1 in the previous example. Moreover, the evaluation of c_2 when $\langle b, \sigma \rangle \to \mathbf{false}$ is also analogous to that of the command c_2 in Example 3.6. Therefore we focus on the analysis of c_2 for the case $\langle b, \sigma \rangle \to \mathbf{true}$. Trivially:

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else\ skip}, \sigma \rangle \to \sigma' \nwarrow \langle b, \sigma \rangle \to \mathbf{true}, \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma' \nwarrow \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma'$$

So we reduce to a sub-goal identical to the one in the evaluation of c_1, and we can conclude that $c_1 \sim c_2$.

3.3.3 Examples: Inequality Proofs

The next example deals with programs that can behave the same or exhibit different behaviours depending on the initial memory.

Example 3.8 (Inequality proof). Let us consider the commands

$$c_1 \stackrel{\text{def}}{=} (\mathbf{while}\ x > 0\ \mathbf{do}\ x := 1); x := 0$$
$$c_2 \stackrel{\text{def}}{=} x := 0$$

Let us prove that $c_1 \not\sim c_2$.

For c_2 we have

$$\begin{array}{rl} \langle x := 0, \sigma\rangle \to \sigma' & \nwarrow_{\sigma' = \sigma[^n/_x]} \ \langle 0, \sigma\rangle \to n \\ & \nwarrow_{n=0} \ \Box \end{array}$$

That is: $\forall \sigma.\ \langle x := 0, \sigma\rangle \to \sigma[^0/_x]$.
Next, we focus on the first part of c_1:

$$w \stackrel{\text{def}}{=} \textbf{while}\ x > 0\ \textbf{do}\ x := 1$$

If $\sigma(x) \leq 0$ it is immediate to check that

$$\langle \textbf{while}\ x > 0\ \textbf{do}\ x := 1, \sigma\rangle \to \sigma$$

The derivation is sketched below:

$$\begin{array}{rl} \langle w, \sigma\rangle \to \sigma' & \nwarrow_{\sigma' = \sigma} \ \langle x > 0, \sigma\rangle \to \textbf{false} \\ & \nwarrow \ \langle x, \sigma\rangle \to n,\ \langle 0, \sigma\rangle \to m,\ n \leq m \\ & \nwarrow_{n = \sigma(x)} \ \langle 0, \sigma\rangle \to m,\ \sigma(x) \leq m \\ & \nwarrow_{m=0} \ \sigma(x) \leq 0 \\ & \nwarrow \ \Box \end{array}$$

Instead, if we assume $\sigma(x) > 0$, then

$$\begin{array}{rl} \langle w, \sigma\rangle \to \sigma' & \nwarrow \ \langle x > 0, \sigma\rangle \to \textbf{true},\ \langle x := 1, \sigma\rangle \to \sigma'',\ \langle w, \sigma''\rangle \to \sigma' \\ & \nwarrow^* \ \langle x := 1, \sigma\rangle \to \sigma'',\ \langle w, \sigma''\rangle \to \sigma' \\ & \nwarrow^*_{\sigma'' = \sigma[^1/_x]} \ \langle w, \sigma[^1/_x]\rangle \to \sigma' \end{array}$$

Let us continue the derivation for $\langle w, \sigma[^1/_x]\rangle \to \sigma'$:

$$\begin{array}{rl} \langle w, \sigma[^1/_x]\rangle \to \sigma' & \nwarrow \ \langle x > 0, \sigma[^1/_x]\rangle \to \textbf{true},\ \langle x := 1, \sigma[^1/_x]\rangle \to \sigma''',\ \langle w, \sigma'''\rangle \to \sigma' \\ & \nwarrow^* \ \langle x := 1, \sigma[^1/_x]\rangle \to \sigma''',\ \langle w, \sigma'''\rangle \to \sigma' \\ & \nwarrow_{\sigma''' = \sigma[^1/_x]} \ \langle w, \sigma[^1/_x]\rangle \to \sigma' \end{array}$$

Now, note that we got the same sub-goal $\langle w, \sigma[^1/_x]\rangle \to \sigma'$ already inspected: hence it is not possible to conclude the derivation, which will loop.
Summing up all the above we conclude that

$$\forall \sigma, \sigma'.\quad \langle \textbf{while}\ x > 0\ \textbf{do}\ x := 1, \sigma\rangle \to \sigma' \quad \Rightarrow \quad \sigma(x) \leq 0 \wedge \sigma' = \sigma$$

We can now complete the reduction for the whole of c_1 when $\sigma(x) \leq 0$ (the case $\sigma(x) > 0$ is discharged, because we know that there is no derivation).

$$\begin{array}{rl} \langle w; x := 0, \sigma\rangle \to \sigma' \nwarrow & \langle w, \sigma\rangle \to \sigma'',\ \langle x := 0, \sigma''\rangle \to \sigma' \\ \nwarrow^{*}_{\sigma''=\sigma} & \langle x := 0, \sigma\rangle \to \sigma' \\ \nwarrow^{*}_{\sigma'=\sigma[0/x]} & \square \end{array}$$

Therefore the evaluation ends with $\sigma' = \sigma[^0/_x]$.
By comparing c_1 and c_2 we have that

- there are memories on which the two commands behave the same (i.e., when $\sigma(x) \leq 0$):

$$\exists \sigma, \sigma'. \begin{cases} \langle (\textbf{while } x > 0 \textbf{ do } x := 1); x := 0, \sigma\rangle \to \sigma' \\ \langle x := 0, \sigma\rangle \to \sigma' \end{cases}$$

- there are also cases for which the two commands exhibit different behaviours:

$$\exists \sigma, \sigma'. \begin{cases} \langle (\textbf{while } x > 0 \textbf{ do } x := 1); x := 0, \sigma\rangle \not\to \\ \langle x := 0, \sigma\rangle \to \sigma' \end{cases}$$

As an example, take any σ with $\sigma(x) = 1$ and $\sigma' = \sigma[^0/_x]$.

Since we can find pairs (σ, σ') such that c_1 loops and c_2 terminates we have that $c_1 \not\sim c_2$.

Note that in disproving the equivalence we have exploited a standard technique in logic: to show that a universally quantified formula is not valid we can exhibit one counterexample. Formally

$$\neg \forall x.(P(x) \Leftrightarrow Q(x)) = \exists x.(P(x) \wedge \neg Q(x)) \vee (\neg P(x) \wedge Q(x))$$

3.3.4 Examples: Diverging Computations

What happens if the program has infinitely many different looping situations? How should we handle the memories for which this happens?

Let us rephrase the definition of equivalence between commands:

$$\forall \sigma, \sigma' \quad \begin{cases} \langle c_1, \sigma\rangle \to \sigma' \Leftrightarrow \langle c_2, \sigma\rangle \to \sigma' \\ \langle c_1, \sigma\rangle \not\to \quad \Leftrightarrow \langle c_2, \sigma\rangle \not\to \end{cases}$$

Next we see an example where this situation emerges.

Example 3.9 (Proofs of non-termination). Let us consider the commands

$$\begin{array}{l} c_1 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := 1 \\ c_2 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := x + 1 \end{array}$$

Is it true that $c_1 \sim c_2$? On the one hand, note that c_1 can only store 1 in x, whereas c_2 can keep incrementing the value stored in x, so one may be led to suspect that the two commands are not equivalent. On the other hand, we know that when the commands diverge, the values stored in the memory locations are inessential.

As already done in previous examples, let us focus on the possible derivation of c_1 by considering two separate cases that depends of the evaluation of the guard $x > 0$:

Case $\sigma(x) \leq 0$) If $\sigma(x) \leq 0$, we know already from Example 3.8 that $\langle c_1, \sigma \rangle \to \sigma$:

$$\begin{array}{rl} \langle c_1, \sigma \rangle \to \sigma' & \nwarrow_{\sigma' = \sigma} \ \langle x > 0, \sigma \rangle \to \textbf{false} \\ & \nwarrow^* \ \Box \end{array}$$

In this case, the body of the **while** is not executed and the resulting memory is left unchanged. We leave to the reader the details of the analogous derivation of c_2, which behaves the same.

Case $\sigma(x) > 0$) If $\sigma(x) > 0$, we know already from Example 3.8 that $\langle c_1, \sigma \rangle \not\to$ Now we must check whether c_2 diverges too when $\sigma(x) > 0$:

$$\begin{array}{rl} \langle c_2, \sigma \rangle \to \sigma' & \nwarrow \ \langle x > 0, \sigma \rangle \to \textbf{true}, \\ & \quad \langle x := x+1, \sigma \rangle \to \sigma_1, \quad \langle c_2, \sigma_1 \rangle \to \sigma' \\ & \nwarrow^* \ \langle x := x+1, \sigma \rangle \to \sigma_1, \quad \langle c_2, \sigma_1 \rangle \to \sigma' \\ \nwarrow^*_{\sigma_1 = \sigma\left[\sigma(x)+1/x\right]} & \langle c_2, \sigma\left[\sigma(x)+1/x\right] \rangle \to \sigma' \\ & \nwarrow \ \langle x > 0, \sigma\left[\sigma(x)+1/x\right] \rangle \to \textbf{true}, \\ & \quad \langle x := x+1, \sigma\left[\sigma(x)+1/x\right] \rangle \to \sigma_2, \\ & \quad \langle c_2, \sigma_2 \rangle \to \sigma' \\ & \nwarrow^* \ \langle x := x+1, \sigma\left[\sigma(x)+1/x\right] \rangle \to \sigma_2, \\ & \quad \langle c_2, \sigma_2 \rangle \to \sigma' \\ \nwarrow^*_{\sigma_2 = \sigma_1\left[\sigma_1(x)+1/x\right] = \sigma\left[\sigma(x)+2/x\right]} & \langle c_2, \sigma\left[\sigma(x)+2/x\right] \rangle \to \sigma' \\ & \dots \end{array}$$

Now the situation is more subtle: we keep looping, but without crossing the same sub-goal twice, because the memory is updated with a different value for x at each iteration. However, using induction, which will be the subject of Section 4.1.3, we can prove that the derivation will not terminate. Roughly, the idea is the following:

- at step 0, i.e., at the first iteration, the cycle does not terminate;
- if at the ith step the cycle has not terminated yet, then it will not terminate at the $(i+1)$th step, because $x > 0 \Rightarrow x+1 > 0$.

The formal proof would require us to show that at the ith iteration the value stored in the memory at location x will be $\sigma(x)+i$, from which we can conclude that the expression $x>0$ will hold true (since by assumption $\sigma(x)>0$ and thus $\sigma(x)+i>0$). Once the proof is completed, we can conclude that c_2 diverges and therefore $c_1 \sim c_2$. Below we outline a simpler technique to prove non-termination that can be used under some circumstances.

Let us consider the command $w \stackrel{\text{def}}{=} \textbf{while}\ b\ \textbf{do}\ c$. As we have seen in the last example, to prove the non-termination of w we can exploit the induction hypotheses over memory states to define the inference rule below: the idea is that if we can find a set S of memories such that, for any $\sigma' \in S$, the guard b is evaluated to **true** and the execution of c leads to a memory σ'' which is also in S, then we can conclude that w diverges when evaluated in any of the memories $\sigma \in S$.

$$\frac{\sigma \in S \quad \forall \sigma' \in S.\langle b, \sigma' \rangle \to \textbf{true} \quad \forall \sigma' \in S, \forall \sigma''.(\langle c, \sigma' \rangle \to \sigma'' \ \Rightarrow\ \sigma'' \in S)}{\langle w, \sigma \rangle \not\to} \tag{3.20}$$

Note that the property

$$\forall \sigma''.(\langle c, \sigma' \rangle \to \sigma'' \ \Rightarrow\ \sigma'' \in S)$$

is satisfied even when $\langle c, \sigma' \rangle \not\to$, because there is no σ'' such that the left-hand side of the implication holds.

Example 3.10. Let us consider again the command c_2 from Example 3.9:

$$c_2 \stackrel{\text{def}}{=} \textbf{while}\ x>0\ \textbf{do}\ x := x+1.$$

We set $S = \{\sigma \mid \sigma(x) > 0\}$, take $\sigma \in S$ and prove the premises of the rule for divergence to conclude that $\langle w, \sigma \rangle \not\to$.

1. We must show that $\forall \sigma' \in S.\langle x>0, \sigma' \rangle \to \textbf{true}$, which follows by definition of S.
2. We need to prove that $\forall \sigma' \in S, \forall \sigma''.(\langle x := x+1, \sigma' \rangle \to \sigma'' \ \Rightarrow\ \sigma'' \in S)$. Take $\sigma' \in S$, i.e., such that $\sigma'(x) > 0$, and assume $\langle x := x+1, \sigma' \rangle \to \sigma''$. Then it must be that $\sigma'' = \sigma'[{}^{\sigma'(x)+1}/_{x}]$ and we have $\sigma''(x) = \sigma'[{}^{\sigma'(x)+1}/_{x}](x) = \sigma'(x)+1 > 0$ because $\sigma'(x) > 0$ by hypothesis. Hence $\sigma'' \in S$.

Recall that, in general, program termination is semi-decidable (and non-termination possibly non-semi-decidable), so we cannot have a proof technique for demonstrating the convergence or divergence of any program.

Example 3.11 (Collatz's algorithm). Consider the algorithm below, which is known as *Collatz's algorithm*, or also as *Half Or Triple Plus One*

$$d \stackrel{\text{def}}{=} x := y\ ;\ k := 0\ ;\ \textbf{while}\ x>0\ \textbf{do}\ (x := x-2\ ;\ k := k+1)$$
$$c \stackrel{\text{def}}{=} \textbf{while}\ y>1\ \textbf{do}\ (d\ ;\ \textbf{if}\ x=0\ \textbf{then}\ y := k\ \textbf{else}\ y := (3\times y)+1)$$

The command d, when executed in a memory σ with $\sigma(y) > 0$, terminates by producing either a memory σ' with $\sigma'(x) = 0$ and $\sigma(y) = 2 \times \sigma'(k)$ (when $\sigma(y)$ is even), or a memory σ'' with $\sigma''(x) = -1$ (when $\sigma(y)$ is odd). The command c exploits d to update at each iteration the value of y to either half of y (when $\sigma(y)$ is even) or three times y plus one (when $\sigma(y)$ is odd).

If integer division ($/$) and remainder ($\%$) operators were present, the algorithm could be written in the simpler form:

$$\begin{array}{l}\textbf{while } y > 1 \textbf{ do } (\\ \qquad \textbf{if } y\%2 = 0 \textbf{ then } y := y/2;\\ \qquad\qquad\qquad \textbf{else } y := (3 \times y) + 1\\ )\end{array}$$

It is an open mathematical conjecture to prove that the command c terminates when executed in any memory σ. The conjecture has been checked by computers and proved true[1] for all starting values of y up to 5×2^{60}.

Problems

3.1. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do } (r := r \times x \ ; \ y := y - 1)$$

Let $c \stackrel{\text{def}}{=} (r := 1 \ ; \ w)$ and $\sigma \stackrel{\text{def}}{=} [^9/_x, ^2/_y]$. Use goal-oriented derivation, according to the operational semantics of IMP, to find the memory σ' such that $\langle c, \sigma \rangle \to \sigma'$, if it exists.

3.2. Consider the IMP command

$$\begin{array}{l} w \stackrel{\text{def}}{=} \textbf{while } y \geq 0 \textbf{ do}\\ \qquad\qquad \textbf{if } y = 0 \textbf{ then } y := y + 1 \textbf{ else skip}\end{array}$$

For which memories σ, σ' do we have $\langle w, \sigma \rangle \to \sigma'$?

3.3. Prove that for any $b \in Bexp, c \in Com$ we have $c \sim \textbf{if } b \textbf{ then } c \textbf{ else } c$.

3.4. Prove that for any $b \in Bexp, c \in Com$ we have $c_1 \sim c_2$, where

$$\begin{array}{l} c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c\\ c_2 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do}\\ \qquad\qquad\qquad \textbf{if } b \textbf{ then } c \textbf{ else skip}\end{array}$$

[1] Source `http://en.wikipedia.org/wiki/Collatz_conjecture`, last visited July 2015.

3.5. Let

$$c_1 \stackrel{\text{def}}{=} c \,;\, \textbf{while}\ b\ \textbf{do}\ c$$
$$c_2 \stackrel{\text{def}}{=} (\textbf{while}\ b\ \textbf{do}\ c) \,;\, c$$

Is it the case that for any $b \in Bexp, c \in Com$ we have $c_1 \sim c_2$?

3.6. Prove that $c_1 \not\sim c_2$, where

$$c_1 \stackrel{\text{def}}{=} \textbf{while}\ x > 0\ \textbf{do}\ x := 0$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while}\ x \geq 0\ \textbf{do}\ x := 0$$

3.7. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while}\ x \leq y\ \textbf{do}\ (x := x+1 \,;\, y := y+2)$$

Find the largest set S of memories such that the command w diverges. Use the inference rule for divergence to prove non-termination.

3.8. Prove that $c_1 \not\sim c_2$, where

$$c_1 \stackrel{\text{def}}{=} \textbf{while}\ x > 0\ \textbf{do}\ x := x+1$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while}\ x \geq 0\ \textbf{do}\ x := x+2$$

3.9. Suppose we extend IMP with the arithmetic expression a_0/a_1 for integer division, whose operational semantics is

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0/a_1, \sigma\rangle \to n}\ n_0 = n_1 \times n\ (\text{div}) \tag{3.21}$$

1. Prove that the semantics of extended arithmetic expressions is not deterministic. In other words, give a counterexample to the property below:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \forall n, m \in \mathbb{Z}.\ (\langle a, \sigma\rangle \to n \wedge \langle a, \sigma\rangle \to m \quad \Rightarrow \quad n = m)$$

2. Prove that the semantics of extended arithmetic expressions is not always defined. In other words, give a counterexample to the property below:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \exists n \in \mathbb{Z}.\ \langle a, \sigma\rangle \to n$$

3.10. Define a small-step operational semantics for IMP. To this aim, introduce a special symbol $\star$ as a termination marker and consider judgements of either the form $\langle c, \sigma\rangle \to \langle c', \sigma'\rangle$ or $\langle c, \sigma\rangle \to \langle \star, \sigma'\rangle$. Define the semantics in such a way that the evaluation is deterministic and that $\langle c, \sigma\rangle \to^* \langle \star, \sigma'\rangle$ if and only if $\langle c, \sigma\rangle \to \sigma'$ in the usual big-step semantics for IMP.

Chapter 4
Induction and Recursion

To understand recursion, you must first understand recursion.
(traditional joke)

Abstract In this chapter we present some induction techniques that will turn out to be useful for proving formal properties of the languages and models presented in the book. We start by introducing Noether's principle of well-founded induction, from which we then derive induction principles over natural numbers, terms of a signature and derivations in a logical system. The chapter ends by presenting well-founded recursion.

4.1 Noether's Principle of Well-Founded Induction

In the literature several different kinds of induction are defined, but they all rely on the so-called *Noether's principle of well-founded induction*. We start by defining this important principle and will then derive several induction methods.

4.1.1 Well-Founded Relations

We recall some key mathematical notions and definitions.

Definition 4.1 (Binary relation). A *binary relation* (*relation* for short) $\prec$ over a set A is a subset of the cartesian product $A \times A$:

$$\prec \subseteq A \times A$$

For $(a,b) \in \prec$ we use the infix notation $a \prec b$ and also write equivalently $b \succ a$. Moreover, we write $a \not\prec b$ in place of $(a,b) \notin \prec$.

A relation $\prec \subseteq A \times A$ can be conveniently represented as an *oriented graph* whose nodes are the elements of A and whose arcs $n \to m$ represent the pairs $(n,m) \in \prec$ in the relation.

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_4

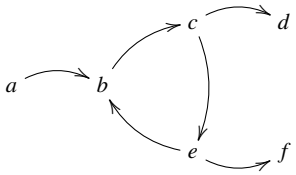


Fig. 4.1: Graph of a relation

Example 4.1. Let $A = \{a, b, c, d, e, f\}$. The graph in Figure 4.1 represents the relation $\prec$ over the set A with $a \prec b$, $b \prec c$, $c \prec d$, $c \prec e$, $e \prec f$, $e \prec b$.

Definition 4.2 (Infinite descending chain). Given a relation $\prec$ over the set A, an *infinite descending chain* is an infinite sequence $\{a_i\}_{i \in \mathbb{N}}$ of elements in A such that

$$\forall i \in \mathbb{N}.\ a_{i+1} \prec a_i$$

An infinite descending chain can be represented as a function a from $\mathbb{N}$ to A such that $a(i)$ decreases (according to $\prec$) as i grows:

$$a(0) \succ a(1) \succ a(2) \succ \cdots$$

Definition 4.3 (Well-founded relation). A relation is *well-founded* if it has no infinite descending chains.

Definition 4.4 (Transitive closure). Let $\prec$ be a relation over A. The *transitive closure* of $\prec$, written $\prec^+$, is defined by the following inference rules:

$$\frac{a \prec b}{a \prec^+ b} \qquad \frac{a \prec^+ b \qquad b \prec^+ c}{a \prec^+ c}$$

By the first rule, $\prec$ is always included in $\prec^+$. It can be proved that $(\prec^+)^+$ always coincides with $\prec^+$.

Definition 4.5 (Transitive and reflexive closure). Let $\prec$ be a relation over A. The *transitive and reflexive closure* of $\prec$, written $\prec^*$, is defined by the following inference rules:

$$\frac{}{a \prec^* a} \qquad \frac{a \prec b}{a \prec^* b} \qquad \frac{a \prec^* b \qquad b \prec^* c}{a \prec^* c}$$

It can be proved that both $\prec$ and $\prec^+$ are included in $\prec^*$ and that $(\prec^*)^*$ always coincides with $\prec^*$.

Example 4.2. Consider the usual "less than" relation $<$ over integers. Since we have, e.g., the infinite descending chain

$$4 > 2 > 0 > -2 > -4 > \cdots$$

it is not well-founded. Note that its transitive closure $<^+$ is the same as $<$.

Example 4.3. Consider the usual "less than" relation $<$ over natural numbers. We cannot have an infinite descending chain $\{a_i\}_{i\in\mathbb{N}}$ because there are only finitely many elements less than a_0. Hence the relation is well-founded. Note that $<^+ = <$.

Example 4.4. Consider the usual "less than or equal to" $\leq$ relation over natural numbers. Since we have, e.g., the infinite descending chain

$$4 \geq 2 \geq 0 \geq 0 \geq 0 \geq \cdots$$

the relation $\leq$ is not well-founded. Note also, than any infinite descending chain must include only a finite number of elements, because there are only a finite number of elements less than or equal to a_0, and therefore there exists some $k \in \mathbb{N}$ such that $\forall i \geq k.\ a_i = a_k$. Note that $\leq$ is the reflexive and transitive closure of $<$, i.e., $<^* = \leq$.

Theorem 4.1. *Let $\prec$ be a relation over A. For any $x, y \in A$, $x \prec^+ y$ if and only if there exist a finite number of elements $z_0, z_1, ..., z_k \in A$ with $k > 0$ such that*

$$x = z_0 \prec z_1 \prec \cdots \prec z_k = y.$$

The proof of the above theorem is left as an exercise (see Problem 4.4).

With respect to the oriented graph associated with the relation $\prec$, we note that $a \prec^+ b$ means that there is a non-empty finite path from a to b, while $a \prec^* b$ means that there is a (possibly empty) finite path from a to b.

Theorem 4.2 (Well-foundedness of $\prec^+$). *A relation $\prec$ is well-founded if and only if its transitive closure $\prec^+$ is well-founded.*

Proof. One implication is trivial: if $\prec^+$ is well-founded then $\prec$ is obviously well-founded, because any descending chain for $\prec$ is also a descending chain for $\prec^+$ (and all such chains are finite by hypothesis).

For the other direction, by contraposition let us assume $\prec^+$ is not well-founded and prove that $\prec$ is not well-founded. Take any infinite descending chain

$$a_0 \succ^+ a_1 \succ^+ a_2 \succ^+ \cdots$$

But whenever $a_i \succ^+ a_{i+1}$, by Theorem 4.1, there is a finite descending $\prec$-chain of elements between a_i and a_{i+1} and therefore we can build an infinite descending chain

$$a_0 \succ \cdots \succ a_1 \succ \cdots \succ a_2 \succ \cdots$$

so that $\prec$ is not well-founded. □

Example 4.5. Consider the "immediate precedence" relation $\prec$ over natural numbers, such that $n \prec n+1$ for all $n \in \mathbb{N}$. Note that the transitive closure of $\prec$ is the usual "less than" relation $<$ over natural numbers, i.e., $\prec^+ = <$. By Theorem 4.2 and Example 4.3 the relation $\prec$ over $\mathbb{N}$ is well-founded.

Definition 4.6 (Acyclic relation). We say that $\prec$ has a cycle if $\exists a \in A.\ a \prec^+ a$. We say that $\prec$ is *acyclic* if it has no cycle (i.e., $\forall a \in A.\ a \not\prec^+ a$).

Theorem 4.3 (Well-founded relations are acyclic). *If the relation $\prec$ is well-founded, then it is acyclic.*

Proof. By contraposition, we prove that if the relation $\prec$ is not acyclic then it is not well-founded. Let us assume that there is $a \in A$ such that $a \prec^+ a$. Then we have a trivial infinite descending chain

$$a \succ^+ a \succ^+ a \succ^+ a \succ^+ \cdots$$

By Theorem 4.2, $\prec$ is not well-founded because $\prec^+$ is not well-founded. □

For example, the relation in Figure 4.1 is not acyclic and thus it is not well-founded.

Theorem 4.4 (Well-founded relations over finite sets). *Let A be a finite set and let $\prec$ be acyclic, then $\prec$ is well-founded.*

Proof. We prove that if $\prec$ is not well-founded then it is not acyclic. Assume is not well-founded, then it has an infinite descending chain. Since A is finite, any descending chain with more than $|A|$ elements must contain (at least) two occurrences, say at positions i and j with $i < j$, of the same element (by the so-called "pigeonhole principle"), so that

$$a_i \succ a_{i+1} \succ \cdots \succ a_{j-1} \succ a_j = a_i$$

forms a cycle. □

Definition 4.7 (Minimal element). Let $\prec$ be a relation over the set A. Given a set $Q \subseteq A$, we say that $m \in Q$ is *minimal* if there is no element $x \in Q$ such that $x \prec m$, i.e., $\forall x \in Q.\ x \not\prec m$.

It follows that Q has no minimal element if $\forall m \in Q.\ \exists x \in Q.\ x \prec m$.

Lemma 4.1 (Well-founded relation). *Let $\prec$ be a relation over the set A. The relation $\prec$ is well-founded if and only if every nonempty subset $Q \subseteq A$ contains a minimal element m.*

Proof. Since $P \Leftrightarrow Q$ is equivalent to $\neg P \Leftrightarrow \neg Q$, the statement of this lemma can be rephrased by saying that *the relation $\prec$ has an infinite descending chain if and only if there exists a nonempty subset $Q \subseteq A$ with no minimal element*.

We prove each implication (of the transformed statement) separately.

$\Rightarrow$) We assume that $\prec$ has an infinite descending chain $a_1 \succ a_2 \succ a_3 \succ \cdots$ and we let $Q = \{a_1, a_2, a_3, \ldots\}$ be the set of all the elements in the infinite descending chain. The set Q has no minimal element, because for any candidate $a_i \in Q$ we know there is one element $a_{i+1} \in Q$ with $a_i \succ a_{i+1}$.

$\Leftarrow$) Let Q be a nonempty subset of A with no minimal element. Since Q is nonempty, it must contain at least one element. We randomly pick an element $a_0 \in Q$. Since a_0 is not minimal there must exist an element $a_1 \in Q$ such that $a_0 \succ a_1$, and we can iterate the reasoning (i.e. a_1 is not minimal and there is $a_2 \in Q$ with $a_0 \succ a_1 \succ a_2$, etc.). So we can build an infinite descending chain. □

Example 4.6 (Natural numbers). Both $n \prec n+1$ (the immediate precedence relation) and $n < n+1+k$ (the usual "less than" relation), with $n,k \in \mathbb{N}$, are simple examples of well-founded relations. In fact, from every element $n \in \mathbb{N}$ we can start a descending chain of length at most n.

Definition 4.8 (Terms over one-sorted signatures). Let $\Sigma = \{\Sigma_n\}_{n\in\mathbb{N}}$ be a one-sorted signature, i.e., a set of ranked operators f such that $f \in \Sigma_n$ if f takes n arguments. We define the set of Σ-terms as the set T_Σ that is defined inductively by the following inference rule:

$$\frac{t_i \in T_\Sigma \quad i = 1,\ldots,n \quad f \in \Sigma_n}{f(t_1,\ldots,t_n) \in T_\Sigma} \tag{4.1}$$

Definition 4.9 (Terms over many-sorted signatures). Let

- S be a set of *sorts* (i.e. the set of the different data types we want to consider);
- $\Sigma = \{\Sigma_{s_1\ldots s_n,s}\}_{s_1,\ldots,s_n,s\in S}$ be a *signature* over S, i.e. a set of typed operators ($f \in \Sigma_{s_1\ldots s_n,s}$ is an operator that takes n arguments, the ith argument being of type s_i, and gives a result of type s).

  We define the set of Σ-*terms* as the set

$$T_\Sigma = \{T_{\Sigma,s}\}_{s\in S}$$

where, for $s \in S$, the set $T_{\Sigma,s}$ is the set of terms of sort s over the signature Σ, defined inductively by the following inference rule:

$$\frac{t_i \in T_{\Sigma,s_i} \quad i = 1,\ldots,n \quad f \in \Sigma_{s_1\ldots s_n,s}}{f(t_1,\ldots,t_n) \in T_{\Sigma,s}}$$

(When S is a singleton, we are in the same situation as in Definition 4.8 and write just Σ_n instead of $\Sigma_{w,s}$ with $w = \underbrace{s \ldots s}_{n}$.)

Since the operators of the signature are known, we can specialise the above rule 4.1 for each operator, i.e. we can consider the set of inference rules

$$\left\{ \frac{t_i \in T_{\Sigma,s_i} \quad i = 1,\ldots,n}{f(t_1,\ldots,t_n) \in T_{\Sigma,s}} \right\}_{f\in\Sigma_{s_1\ldots s_n,s}} \tag{4.2}$$

Note that, as a special case of the above inference rule, for constants $a \in \Sigma_{\varepsilon,s}$ we have

$$\frac{}{a \in T_{\Sigma,s}} \tag{4.3}$$

Example 4.7 (IMP signature). In the case of IMP, we have $S = \{Aexp, Bexp, Com\}$ and then we have an operation for each production in the grammar.

For example, the sequential composition of commands ";" corresponds to the binary infix operator $(-;-) \in \Sigma_{ComCom,Com}$.

Similarly the boolean expression that tests for equality is built using the binary infix operator $(- = -) \in \Sigma_{AexpAexp,Bexp}$.

By abusing the notation, we often write Com for $T_{\Sigma,Com}$ (respectively, $Aexp$ for $T_{\Sigma,Aexp}$ and $Bexp$ for $T_{\Sigma,Bexp}$).

Then, we have inference rule instances such as:

$$\frac{}{skip \in Com} \qquad \frac{skip \in Com \quad x := 1 \in Com}{skip; x := 1 \in Com}$$

The programs we consider are (well-formed) terms over a suitable signature Σ (possibly many-sorted). Therefore it is useful to define a well-founded containment relation between a term and its subterms. For example, we will exploit this relation when dealing with structural induction in Section 4.1.5.

Example 4.8 (Terms and subterms). For any n-ary function symbol $f \in \Sigma_n$ and terms $t_1, \ldots, t_n$, we let

$$t_i \prec f(t_1, \ldots, t_n) \qquad i = 1, \ldots, n$$

The idea is that a term t_i precedes (according to $\prec$, i.e. it is less than) any term that contains it as a subterm (e.g. as an argument).

As a concrete example, let us consider the signature Σ with $\Sigma_0 = \{c\}$ and $\Sigma_2 = \{f\}$. Then, we have, e.g.,

$$c \prec f(c,c) \prec f(f(c,c),c) \prec f(f(f(c,c),c), f(c,c))$$

If we look at terms as trees (function symbols as nodes with one child for each argument and constants as leaves), then we can observe that whenever $s \prec t$ the depth of s is strictly less than the depth of t. Therefore any descending chain is finite (the length is at most the depth of the first term of the chain). Moreover, in the particular case above, c is the only constant and therefore the only minimal element.

Example 4.9 (Lexicographic precedence relation). A quite common (well-founded) relation is the so-called lexicographic precedence relation. The idea is to have elements that are strings over a given ordered alphabet and to compare them symbol by symbol, from the leftmost to the rightmost: as soon as we find a symbol in one string that precedes the symbol in the same position of the other string, then we assume that the former string precedes the latter (independently of the remaining symbols of the two strings).

As a concrete example, let us consider the set of all pairs $\langle n, m \rangle$ of natural numbers ordered by immediate precedence. The lexicographic precedence relation is defined as (see Figure 4.2):

- $\forall n, m, k.\ (\langle n, m \rangle \prec \langle n+1, k \rangle)$
- $\forall n, m.\ (\langle n, m \rangle \prec \langle n, m+1 \rangle)$

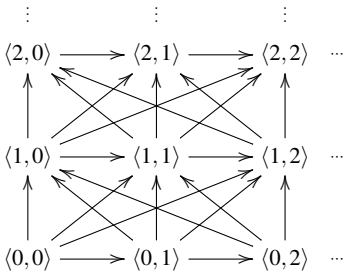


Fig. 4.2: Graph of the lexicographic precedence relation over pairs of natural numbers

By Theorem 4.2, the relation $\prec$ is well-founded if and only if its transitive closure is such. Note that the relation $\prec^+$ has no cycle and any descending chain is bounded by the only minimal element $\langle 0,0\rangle$. For example, we have

$$\langle 5,1\rangle \succ^+ \langle 4,25\rangle \succ^+ \langle 3,100\rangle \succ^+ \langle 3,14\rangle \succ^+ \langle 2,1\rangle \succ^+ \langle 1,1000\rangle \succ^+ \langle 0,0\rangle$$

It is worth noting that any element $\langle n,m\rangle$ with $n \geq 1$ is preceded by infinitely many elements (e.g., $\forall k.\ \langle 0,k\rangle \prec \langle 1,0\rangle$) and it can be the first element of infinitely many (finite) descending chains (of unbounded length).

Still, given any nonempty set $Q \subseteq \mathbb{N} \times \mathbb{N}$, it is easy to find a minimal element $m \in Q$, namely such that $\forall b \prec^+ m.\ b \notin Q$. In fact, we can just take $m = \langle m_1, m_2\rangle$, where m_1 is the minimum (w.r.t. the usual less than relation over natural numbers) of the set $Q_1 = \{n_1 \mid \langle n_1, n_2\rangle \in Q\}$ and m_2 is the minimum of the set $Q_2 = \{n_2 \mid \langle m_1, n_2\rangle \in Q\}$. Note that Q_1 is nonempty because Q is such by hypothesis, and Q_2 is nonempty because $m_1 \in Q_1$ and therefore there must exist at least one pair $\langle m_1, n_2\rangle \in Q$ for some n_2. Thus

$$\langle m_1 = min\{n_1 \mid \langle n_1, n_2\rangle \in Q\}, min\{n_2 \mid \langle m_1, n_2\rangle \in Q\}\rangle$$

is a (the only) minimal element of Q. By Lemma 4.1 the relation $\prec^+$ is well-founded and so is $\prec$ (by Theorem 4.2).

4.1.2 Noetherian Induction

Theorem 4.5. *Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then*

$$(\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a)) \quad \Leftrightarrow \quad \forall a \in A.\ P(a)$$

Proof. We prove the two implications separately:

⇒) We proceed by contraposition. We assume that $\neg(\forall a \in A.\ P(a))$, i.e., that $\exists a \in A.\ \neg P(a)$. Let us consider the nonempty set $Q = \{a \in A \mid \neg P(a)\}$ of all those elements a in A for which $P(a)$ is false. Since $\prec$ is well-founded, we know by Lemma 4.1 that there is a minimal element $m \in Q$. Obviously $\neg P(m)$ (otherwise m cannot be in Q). Since m is minimal in Q, then $\forall b \prec m.\ b \notin Q$, i.e., $\forall b \prec m.\ P(b)$. Then, the element m is a counterexample to the property $(\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a))$, because $\forall b \prec m.\ P(b)$ but $\neg P(m)$. Hence we conclude that

$$\neg(\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a))$$

⇐) We observe that if $\forall a.\ P(a)$ then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ is true for any a because the premise $(\forall b \prec a.\ P(b))$ is not relevant (the conclusion of the implication is true). □

From the first implication, the validity of the following induction principle follows.

Definition 4.10 (Noetherian induction). Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then the following inference rule is called *Noetherian induction*.

$$\frac{\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a)}{\forall a \in A.\ P(a)} \tag{4.4}$$

We call a *base case* any element $a \in A$ such that the set of its predecessors $\{b \in A \mid b \prec a\}$ is empty.

4.1.3 Weak Mathematical Induction

The principle of weak mathematical induction is a special case of Noetherian induction that is frequently used to prove formulas over the set of natural numbers: we take

$$A = \mathbb{N} \qquad n \prec m \Leftrightarrow m = n+1$$

In this case

- if we take $a = 0$ then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $P(0)$, because there is no $b \in \mathbb{N}$ such that $b \prec 0$;
- if we take $a = n+1$ for some $n \in \mathbb{N}$, then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $P(n) \Rightarrow P(n+1)$.

In other words, to prove that $P(n)$ holds for all $n \in \mathbb{N}$ we can just prove that

- $P(0)$ holds (base case), and
- given a generic $n \in \mathbb{N}$, $P(n+1)$ holds whenever $P(n)$ holds (inductive case).

Definition 4.11 (Weak mathematical induction).

$$\frac{P(0) \qquad \forall n \in \mathbb{N}.\ (P(n) \Rightarrow P(n+1))}{\forall n \in \mathbb{N}.\ P(n)} \tag{4.5}$$

The weak mathematical induction principle is helpful, because it allows us to exploit the hypothesis $P(n)$ when proving $P(n+1)$.

4.1.4 Strong Mathematical Induction

The principle of strong mathematical induction extends the weak one by strengthening the hypotheses under which $P(n+1)$ is proved to hold. We take

$$A = \mathbb{N} \qquad n \prec m \ \Leftrightarrow\ \exists k \in \mathbb{N}.\ m = n+k+1$$

In this case

- if we take $a = 0$ then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $P(0)$, as for the case of weak mathematical induction;
- if we take $a = n+1$ for some $n \in \mathbb{N}$, then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $(P(0) \wedge P(1) \wedge \cdots \wedge P(n)) \Rightarrow P(n+1)$, i.e., using a more concise notation, $(\forall i \leq n.\ P(i)) \Rightarrow P(n+1)$.

In other words, to prove that $P(n)$ holds for any $n \in \mathbb{N}$ we can just prove that

- $P(0)$ holds, and
- given a generic $n \in \mathbb{N}$, $P(n+1)$ holds whenever $P(i)$ holds for all $i = 0, ..., n$.

Definition 4.12 (Strong mathematical induction).

$$\frac{P(0) \qquad \forall n \in \mathbb{N}.\ (\forall i \leq n.\ P(i)) \Rightarrow P(n+1)}{\forall n \in \mathbb{N}.\ P(n)} \tag{4.6}$$

The adjective "strong" comes from the fact that to prove $P(n+1)$ we can now exploit the *stronger* hypothesis $P(0) \wedge P(1) \wedge ... \wedge P(n)$ instead of just $P(n)$.

4.1.5 Structural Induction

The principle of structural induction is a special instance of Noetherian induction for proving properties over the set of terms generated by a given signature. Here, the precedence relation binds a term to its subterms.

Structural induction takes T_Σ as the set of elements and the subterm-term relation as the well-founded relation

$$A = T_\Sigma \qquad t_i < f(t_1, \ldots, t_n) \quad i = 1, \ldots, n$$

Definition 4.13 (Structural induction).

$$\frac{\forall t \in T_\Sigma.\ (\forall t' < t.\ P(t')) \Rightarrow P(t)}{\forall t \in T_\Sigma.\ P(t)} \tag{4.7}$$

By exploiting the definition of the well-founded subterm relation, we can expand the above principle as the rule

$$\frac{\forall f \in \Sigma_{s_1 \ldots s_n, s}.\ \forall t_1 \in T_{\Sigma, s_1} \ldots \forall t_n \in T_{\Sigma, s_n}.\ (P(t_1) \wedge \ldots \wedge P(t_n)) \Rightarrow P(f(t_1, \ldots, t_n))}{\forall t \in T_\Sigma.\ P(t)}$$

An easy link can be established with mathematical induction by taking a unique sort, a constant 0 and a unary operation *succ* (i.e., $\Sigma = \Sigma_0 \cup \Sigma_1$ with $\Sigma_0 = \{0\}$ and $\Sigma_1 = \{succ\}$). Then, the structural induction rule would become

$$\frac{P(0) \quad \forall t.\ (P(t) \Rightarrow P(succ(t)))}{\forall t.\ P(t)}$$

Example 4.10. Let us consider the grammar of IMP arithmetic expressions:

$$a \quad ::= \quad n \quad | \quad x \quad | \quad a_0 + a_1 \quad | \quad a_0 - a_1 \quad | \quad a_0 \times a_1$$

How do we exploit structural induction to prove that a property $P(\cdot)$ holds for all arithmetic expressions a? (Namely, we want to prove that $\forall a \in Aexp.\ P(a)$.) The structural induction rule is

$$\frac{\begin{array}{c}\forall n.\ P(n) \qquad \forall x.\ P(x) \qquad \forall a_0, a_1.\ (P(a_0) \wedge P(a_1) \Rightarrow P(a_0 + a_1)) \\ \forall a_0, a_1.\ (P(a_0) \wedge P(a_1) \Rightarrow P(a_0 - a_1)) \qquad \forall a_0, a_1.\ (P(a_0) \wedge P(a_1) \Rightarrow P(a_0 \times a_1))\end{array}}{\forall a.\ P(a)}$$

Essentially, to prove that $\forall a \in Aexp.\ P(a)$, we just need to show that the property holds for any production, i.e., we need to prove that all of the following hold

- $P(n)$ holds for any integer n;
- $P(x)$ holds for any identifier x;
- $P(a_0 + a_1)$ holds whenever both $P(a_0)$ and $P(a_1)$ hold;
- $P(a_0 - a_1)$ holds whenever both $P(a_0)$ and $P(a_1)$ hold;
- $P(a_0 \times a_1)$ holds whenever both $P(a_0)$ and $P(a_1)$ hold.

Example 4.11 (Termination of arithmetic expressions). Let us consider the case of arithmetic expressions seen above and prove that the evaluation of expressions always terminates (a property that is also called *normalisation*):[1]

$$\forall a \in Aexp.\ \forall \sigma \in \Sigma.\ \exists m \in \mathbb{Z}.\ \langle a, \sigma \rangle \to m$$

[1] We recall that the (overloaded) symbol Σ stands here for the set of memories and not for a generic signature.

In this case we let

$$P(a) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m \in \mathbb{Z}.\ \langle a, \sigma \rangle \to m$$

We prove that $\forall a \in Aexp.\ P(a)$ by structural induction. This amounts to proving that

- $P(n) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m \in \mathbb{Z}.\ \langle n, \sigma \rangle \to m$ holds for any integer n. Trivially, by applying rule (num) we take $m = n$ and we are done.
- $P(x) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m \in \mathbb{Z}.\ \langle x, \sigma \rangle \to m$ holds for any location x. Trivially, by applying rule (ide) we take $m = \sigma(x)$ and we are done.
- $P(a_0) \wedge P(a_1) \Rightarrow P(a_0 + a_1)$ for any arithmetic expressions a_0 and a_1. We assume

$$P(a_0) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m_0 \in \mathbb{Z}.\ \langle a_0, \sigma \rangle \to m_0$$
$$P(a_1) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m_1 \in \mathbb{Z}.\ \langle a_1, \sigma \rangle \to m_1$$

  We want to prove that

$$P(a_0 + a_1) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m \in \mathbb{Z}.\ \langle a_0 + a_1, \sigma \rangle \to m$$

  Take a generic $\sigma \in \Sigma$. We want to find $m \in \mathbb{Z}$ such that $\langle a_0 + a_1, \sigma \rangle \to m$. By applying rule (sum) we can take $m = m_0 + m_1$ if we prove that $\langle a_0, \sigma \rangle \to m_0$ and $\langle a_1, \sigma \rangle \to m_1$. But by the inductive hypothesis we know that such m_0 and m_1 exist and we are done.
- $P(a_0) \wedge P(a_1) \Rightarrow P(a_0 - a_1)$ for any arithmetic expressions a_0 and a_1. The proof is analogous to the previous case and thus omitted.
- $P(a_0) \wedge P(a_1) \Rightarrow P(a_0 \times a_1)$ for any arithmetic expressions a_0 and a_1. The proof is analogous to the previous case and thus omitted.

Example 4.12 (Determinism of arithmetic expressions). Let us consider again the case of IMP arithmetic expressions and prove that their evaluation is deterministic:

$$\forall a \in Aexp.\ \forall \sigma \in \Sigma.\ \forall m, m' \in \mathbb{Z}.\ (\langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m') \Rightarrow m = m'$$

In other words, we want to show that given any arithmetic expression a and any memory σ the evaluation of a in σ will always return exactly one value. We let

$$P(a) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \forall m, m' \in \mathbb{Z}.\ (\langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m') \Rightarrow m = m'$$

We proceed by structural induction.

$a = n$) We want to prove that

$$P(n) \stackrel{\text{def}}{=} \forall \sigma, m, m'.\ (\langle n, \sigma \rangle \to m \wedge \langle n, \sigma \rangle \to m') \Rightarrow m = m'$$

holds. Let us take generic σ, m, m'. We assume the premises $\langle n, \sigma \rangle \to m$ and $\langle n, \sigma \rangle \to m'$ and prove that $m = m'$. In fact, there is only one

rule that can be used to evaluate an integer number, and it always returns the same value. Therefore $m = n = m'$.

$a = x$) We want to prove that

$$P(x) \stackrel{\text{def}}{=} \forall \sigma, m, m'.\ (\langle x, \sigma \rangle \to m \wedge \langle x, \sigma \rangle \to m') \Rightarrow m = m'$$

holds. We assume the premises $\langle x, \sigma \rangle \to m$ and $\langle x, \sigma \rangle \to m'$ and prove that $m = m'$. Again, there is only one rule that can be applied, whose outcome depends on σ. Since σ is the same in both cases, $m = \sigma(x) = m'$.

$a = a_0 + a_1$) We assume the inductive hypotheses

$$P(a_0) \stackrel{\text{def}}{=} \forall \sigma, m_0, m_0'.\ (\langle a_0, \sigma \rangle \to m_0 \wedge \langle a_0, \sigma \rangle \to m_0') \Rightarrow m_0 = m_0'$$
$$P(a_1) \stackrel{\text{def}}{=} \forall \sigma, m_1, m_1'.\ (\langle a_1, \sigma \rangle \to m_1 \wedge \langle a_1, \sigma \rangle \to m_1') \Rightarrow m_1 = m_1'$$

and we want to prove that $P(a_0 + a_1)$, i.e., that

$$\forall \sigma, m, m'.\ (\langle a_0 + a_1, \sigma \rangle \to m \wedge \langle a_0 + a_1, \sigma \rangle \to m') \Rightarrow m = m'$$

We assume the premises $\langle a_0 + a_1, \sigma \rangle \to m$ and $\langle a_0 + a_1, \sigma \rangle \to m'$ and prove that $m = m'$. By the first premise, it must be that $m = m_0 + m_1$ for some m_0, m_1 such that $\langle a_0, \sigma \rangle \to m_0$ and $\langle a_1, \sigma \rangle \to m_1$, because there is only one rule applicable to $a_0 + a_1$; analogously, by the second premise, we must have $m' = m_0' + m_1'$ for some m_0', m_1' such that $\langle a_0, \sigma \rangle \to m_0'$ and $\langle a_1, \sigma \rangle \to m_1'$. By the inductive hypothesis $P(a_0)$ we know that $m_0 = m_0'$ and by $P(a_1)$ we have $m_1 = m_1'$. Thus, $m = m_0 + m_1 = m_0' + m_1' = m'$ and thus $P(a_0 + a_1)$ holds.

The remaining cases for $a = a_0 - a_1$ and $a = a_0 \times a_1$ follow exactly the same pattern as that of $a = a_0 + a_1$.

4.1.6 Induction on Derivations

We can define an induction principle over the set of derivations of a logical system. See Definitions 2.1 and 2.4 for the notions of inference rule and derivation.

Definition 4.14 (Immediate sub-derivation). We say that d' is an *immediate sub-derivation* of d, or simply a sub-derivation of d, written $d' \prec d$, if and only if d has the form $(\{d_1, ..., d_n\} \ / \ y)$ with $d_1 \Vdash_R x_1, ..., d_n \Vdash_R x_n$ and $(\{x_1, ..., x_n\} \ / \ y) \in R$ (i.e., $d \Vdash_R y$) and $d' = d_i$ for some $1 \leq i \leq n$.

Example 4.13 (Immediate sub-derivation). Let us consider the derivation

$$\dfrac{\dfrac{}{\langle 1,\sigma\rangle \to 1}\,\text{num} \qquad \dfrac{}{\langle 2,\sigma\rangle \to 2}\,\text{num}}{\langle 1+2,\sigma\rangle \to 1+2=3}\,\text{sum}$$

The two derivations

$$\dfrac{}{\langle 1,\sigma\rangle \to 1}\,\text{num} \qquad \dfrac{}{\langle 2,\sigma\rangle \to 2}\,\text{num}$$

are immediate sub-derivations of the derivation that exploits rule (sum).

We can derive the notion of proper sub-derivations from that of immediate ones.

Definition 4.15 (Proper sub-derivation). We say that d' is a *proper sub-derivation* of d if and only if $d' \prec^+ d$.

Note that both $\prec$ and $\prec^+$ are well-founded, so they can be used in proofs by induction.

For example, the induction principle based on immediate sub-derivations can be phrased as follows.

Definition 4.16 (Induction on derivations). Let R be a set of inference rules and D the set of derivations defined on R, then

$$\dfrac{\forall \{x_1,\ldots,x_n\}/y \in R.\ (\forall d_i \Vdash_R x_i.\ P(d_1)\wedge\ldots\wedge P(d_n)) \Rightarrow P(\{d_1,\ldots,d_n\}/y)}{\forall d \in D.\ P(d)} \tag{4.8}$$

(Note that $d_1,\ldots,d_n$ are derivations for $x_1,\ldots,x_n$, respectively).

Induction on derivations shares similarities with structural induction. The analogy comes from viewing the (instances of) inference rules as symbolic operators to construct derivations, with axioms playing the role of constants.

4.1.7 Rule Induction

The last kind of induction principle we shall consider applies to sets of elements that are defined by means of inference rules: we have a set of inference rules that establish which elements belong to the set (i.e. the theorems of the logical system) and we need to prove that the application of any such rule will not compromise the validity of a given predicate.

Recall that a rule has the form $(\varnothing/y)$ if it is an axiom, or $(\{x_1,\ldots,x_n\}/y)$ otherwise. Given a set R of such rules, the *set of theorems of* R is defined as

$$I_R = \{y \mid \Vdash_R y\}$$

The rule induction principle states that to show that the property P holds for all elements of I_R, we can prove the following:

- $P(y)$ holds for any axiom $\varnothing/y \in R$;
- for any other rule $\{x_1,\ldots,x_n\}/y \in R$ we have $(\forall 1 \le i \le n.\ x_i \in I_R \wedge P(x_i)) \Rightarrow P(y)$.

Definition 4.17 (Rule induction). Let R be a logical system. The principle of rule induction is

$$\frac{\forall (X/y) \in R.\ (X \subseteq I_R \ \wedge\ \forall x \in X.\ P(x)) \Rightarrow P(y)}{\forall x \in I_R.\ P(x)} \tag{4.9}$$

The principle of rule induction is a useful variant of induction on derivations. In fact by assuming that $X \subseteq I_R$ it follows that there is a derivation d_i for each theorem $x_i \in X$, so that a longer derivation for y is built by applying the rule $(X/y) \in R$ to $d_1,\ldots,d_n$.

Note that in many cases we will use the simpler but less powerful rule

$$\frac{\forall (X/y) \in R.\ (\forall x \in X.\ P(x)) \Rightarrow P(y)}{\forall x \in I_R.\ P(x)} \tag{4.10}$$

In fact, if the latter applies, also the former does, since the implication in the premise must be proved in fewer cases: only for rules X/y such that all the formulas in X are theorems. However, usually it is difficult to take advantage of the restriction.

Example 4.14 (Determinism of IMP commands). We have seen in Example 4.12 that structural induction can be conveniently used to prove that the evaluation of arithmetic expressions is deterministic. Formally, we were proving the predicate $P(\cdot)$ over arithmetic expressions defined as

$$P(a) \stackrel{\text{def}}{=} \forall \sigma.\forall m,m'.\ \langle a,\sigma\rangle \to m \wedge \langle a,\sigma\rangle \to m' \Rightarrow m = m'$$

While the case of boolean expressions is completely analogous, for commands we cannot use the same proof strategy, because structural induction cannot deal with the rule (whtt). In this example, we show that rule induction provides a convenient strategy to solve the problem.

Let us consider the following predicate over theorems, i.e., statements of the form $\langle c,\sigma\rangle \to \sigma'$:

$$Q(\langle c,\sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1 \in \Sigma.\ \langle c,\sigma\rangle \to \sigma_1 \Rightarrow \sigma' = \sigma_1$$

We proceed by rule induction:

rule skip): we want to show that

$$Q(\langle \mathbf{skip},\sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle \mathbf{skip},\sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma$$

which is obvious because there is only one rule applicable to **skip**:

$$\langle \mathbf{skip},\sigma\rangle \to \sigma_1 \quad \nwarrow_{\sigma_1=\sigma} \quad \square$$

rule assign): assuming

$$\langle a, \sigma \rangle \to m$$

we want to show that

$$Q(\langle x := a, \sigma \rangle \to \sigma[{}^{m}/_{x}]) \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle x := a, \sigma \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma[{}^{m}/_{x}]$$

Let us take a generic memory σ_1 and assume the premise $\langle x := a, \sigma \rangle \to \sigma_1$ of the implication holds. We proceed in a goal oriented way. We have

$$\langle x := a, \sigma \rangle \to \sigma_1 \quad \nwarrow_{\sigma_1 = \sigma[{}^{m'}/_{x}]} \quad \langle a, \sigma \rangle \to m'$$

But we know that the evaluation of arithmetic expressions is deterministic and therefore $m' = m$ and $\sigma_1 = \sigma[{}^{m}/_{x}]$.

rule seq): assuming

$$Q(\langle c_0, \sigma \rangle \to \sigma'') \stackrel{\text{def}}{=} \forall \sigma_1''.\ \langle c_0, \sigma \rangle \to \sigma_1'' \Rightarrow \sigma'' = \sigma_1''$$
$$Q(\langle c_1, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle c_1, \sigma'' \rangle \to \sigma_1 \Rightarrow \sigma' = \sigma_1$$

we want to show that

$$Q(\langle c_0; c_1, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle c_0; c_1, \sigma \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

We assume the premise $\langle c_0; c_1, \sigma \rangle \to \sigma_1$ and prove that $\sigma_1 = \sigma'$. We have

$$\langle c_0; c_1, \sigma \rangle \to \sigma_1 \quad \nwarrow \quad \langle c_0, \sigma \rangle \to \sigma_1'', \quad \langle c_1, \sigma_1'' \rangle \to \sigma_1$$

But now we can apply the first inductive hypothesis:

$$Q(\langle c_0, \sigma \rangle \to \sigma'') \stackrel{\text{def}}{=} \forall \sigma_1''.\ \langle c_0, \sigma \rangle \to \sigma_1'' \Rightarrow \sigma_1'' = \sigma''$$

to conclude that $\sigma_1'' = \sigma''$, which together with the second inductive hypothesis

$$Q(\langle c_1, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle c_1, \sigma'' \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

allows us to conclude that $\sigma_1 = \sigma'$.

rule iftt): assuming

$$\langle b, \sigma \rangle \to \mathbf{true}$$
$$Q(\langle c_0, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle c_0, \sigma \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

we want to show that

$$Q(\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

Since $\langle b, \sigma\rangle \to \textbf{true}$ and the evaluation of boolean expressions is deterministic, we have

$$\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma_1 \quad \nwarrow^* \quad \langle c_0, \sigma\rangle \to \sigma_1$$

But then, exploiting the inductive hypothesis

$$Q(\langle c_0, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle c_0, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

we can conclude that $\sigma_1 = \sigma'$.

rule ifff): omitted (it is analogous to the previous case).

rule whff): assuming

$$\langle b, \sigma\rangle \to \textbf{false}$$

we want to show that

$$Q(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma$$

Since $\langle b, \sigma\rangle \to \textbf{false}$ and the evaluation of boolean expressions is deterministic, we have

$$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma_1 \quad \nwarrow^*_{\sigma_1 = \sigma} \quad \square$$

rule whtt): assuming

$$\langle b, \sigma\rangle \to \textbf{true}$$

$$Q(\langle c, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \forall \sigma_1''.\ \langle c, \sigma\rangle \to \sigma_1'' \Rightarrow \sigma_1'' = \sigma''$$

$$Q(\langle \underbrace{\textbf{while } b \textbf{ do } c}_{w}, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle w, \sigma''\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

we want to show that

$$Q(\langle w, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle w, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

Since $\langle b, \sigma\rangle \to \textbf{true}$ and the evaluation of boolean expressions is deterministic, we have

$$\langle w, \sigma\rangle \to \sigma_1 \quad \nwarrow^* \quad \langle c, \sigma\rangle \to \sigma_1'',\ \langle w, \sigma_1''\rangle \to \sigma_1$$

But now we can apply the first inductive hypothesis:

$$Q(\langle c, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \forall \sigma_1''.\ \langle c, \sigma\rangle \to \sigma_1'' \Rightarrow \sigma_1'' = \sigma''$$

to conclude that $\sigma_1'' = \sigma''$, which together with the second inductive hypothesis

$$Q(\langle w, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle w, \sigma'' \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

allow us to conclude that $\sigma_1 = \sigma'$.

4.2 Well-Founded Recursion

We conclude this chapter by presenting the concept of well-founded recursion. A recursive definition of a function f is *well-founded* when the recursive calls to f take as arguments values that are smaller w.r.t. the ones taken by the defined function (according to a suitable well-founded relation). A special class of functions defined on natural numbers according to the principle of well-founded recursion is that of *primitive recursive functions*.

Definition 4.18 (Primitive recursive functions). The primitive recursive functions are those (n-ary) functions over natural numbers obtained according to (any finite application of) the following rules:

zero: The constant 0 is primitive recursive.

succ.: The successor function $s : \mathbb{N} \to \mathbb{N}$ with $s(n) = n + 1$ is primitive recursive.

proj.: For any $i, k \in \mathbb{N}, 1 \leq i \leq k$, the projection functions $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ with

$$\pi_i^k(n_1, ..., n_k) = n_i$$

are primitive recursive.

comp.: Given a k-ary primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$, and k primitive recursive functions $g_1, ..., g_k : \mathbb{N}^m \to \mathbb{N}$ of arity m, the m-ary function h obtained by composing f with $g_1, ..., g_k$ as shown below is primitive recursive:

$$h(n_1, ..., n_m) \stackrel{\text{def}}{=} f(g_1(n_1, ..., n_m), ..., g_k(n_1, ..., n_m))$$

pr.rec.: Given a k-ary primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$ and a $(k+2)$-ary primitive recursive function $g : \mathbb{N}^{k+2} \to \mathbb{N}$, the $(k+1)$-ary function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined as the primitive recursion of f and g below is primitive recursive:

$$\begin{aligned} h(0, n_1, ..., n_k) &= f(n_1, ..., n_k) \\ h(s(n), n_1, ..., n_k) &= g(n, h(n, n_1, \ldots, n_k), n_1, \ldots, n_k) \end{aligned}$$

Note that $\pi_1^1 : \mathbb{N} \to \mathbb{N}$ is the usual identity function. It can be proved that every primitive recursive function is total and computable.

Example 4.15. Addition can be recursively defined with the rules

$$add(0,m) \stackrel{\text{def}}{=} m$$
$$add(n+1,m) \stackrel{\text{def}}{=} add(n,m)+1$$

This does not fit immediately into the above scheme of primitive recursive functions, but we can rephrase the definition as:

$$add(0,n_1) \stackrel{\text{def}}{=} \pi^1_1(n_1)$$
$$add(s(n),n_1) \stackrel{\text{def}}{=} s(\pi^3_2(n,add(n,n_1),n_1))$$

In the primitive recursive style, *add* plays the role of h, the identity function π^1_1 plays the role of f and the composition of s with π^3_2 plays the role of g (so that it receives the unnecessary arguments n and n_1).

Let us make well-founded recursion more precise.

Definition 4.19 (Set of predecessors). Given a well-founded relation $\prec \subseteq A \times A$, the set of *predecessors* of a set $I \subseteq A$ is the set

$$\prec^{-1} I = \{\, b \in A \mid \exists a \in I.\ b \prec a \,\}$$

For $I \subseteq A$ and $f : A \to B$, we denote by $f \restriction I$ the restriction of f to values in I, i.e., $f \restriction I : I \to B$ and $(f \restriction I)(a) = f(a)$ for any $a \in I$.

Theorem 4.6 (Well-founded recursion). *Let $\prec \subseteq A \times A$ be a well-founded relation over A. Let us consider a function F with $F(a,h) \in B$ for any*

- $a \in A$
- $h : (\prec^{-1}\{a\}) \to B$ *(i.e., h is any function whose domain is the set of predecessors of a and whose codomain is B)*

Then, there exists one and only one function $f : A \to B$ which satisfies the equation

$$\forall a \in A.\ f(a) = F(a, f \restriction (\prec^{-1}\{a\}))$$

Proof. The proof is divided in two parts: 1) we first demonstrate that if such a function f exists, then it is unique; and 2) we prove its existence.

1. Uniqueness follows if we can prove the predicate $\forall a.\ P(a)$, where

$$P(a) \stackrel{\text{def}}{=} (\ \forall y \prec^* a.\ (\ f(y) = F(y, f \restriction (\prec^{-1}\{y\})) \wedge g(y) = F(y, g \restriction (\prec^{-1}\{y\}))\ )\ )$$
$$\Rightarrow\ f(a) = g(a)$$

In fact, suppose there are two functions $f, g : A \to B$ such that

$$\forall a \in A.\ f(a) = F(a, f \restriction (\prec^{-1}\{a\}))$$
$$\forall a \in A.\ g(a) = F(a, g \restriction (\prec^{-1}\{a\}))$$

Clearly, for any $a \in A$ the premise

$$(\ \forall y \prec^* a.\ (\ f(y) = F(y, f \upharpoonright (\prec^{-1}\{y\})) \ \wedge\ g(y) = F(y, g \upharpoonright (\prec^{-1}\{y\}))\ )\ )$$

is true and thus we can conclude $f(a) = g(a)$.
The proof that $\forall a.\ P(a)$ goes by well-founded induction on $\prec$. For $a \in A$, we assume that $\forall b \prec a.\ P(b)$ and we want to prove $P(a)$. Suppose that

$$(\ \forall y \prec^* a.\ (\ f(y) = F(y, f \upharpoonright (\prec^{-1}\{y\})) \ \wedge\ g(y) = F(y, g \upharpoonright (\prec^{-1}\{y\}))\ )\ )$$

We need to prove that $f(a) = g(a)$. For $b \prec a$ we must have $f(b) = g(b)$, because $P(b)$ holds by the inductive hypothesis. Thus

$$f \upharpoonright (\prec^{-1}\{a\}) = g \upharpoonright (\prec^{-1}\{a\})$$

and therefore

$$f(a) = F(a, f \upharpoonright (\prec^{-1}\{a\})) = F(a, g \upharpoonright (\prec^{-1}\{a\})) = g(a)$$

2. For existence, we build a family of functions

$$\left\{ f_a : (\prec^{*-1}\{a\}) \to B \right\}_{a \in A}$$

and then take $f \stackrel{\text{def}}{=} \bigcup_{a \in A} f_a$. The existence of the functions f_a is guaranteed by proving that the following property holds for all $x \in A$:

$$Q(x) \stackrel{\text{def}}{=} \exists f_x : (\prec^{*-1}\{x\}) \to B.\ \forall y \prec^* x.\ f_x(y) = F(y, f_x \upharpoonright (\prec^{-1}\{y\}))$$

The proof goes by well-founded recursion. We assume $\forall b \prec a.\ Q(b)$ and prove that $Q(a)$ holds. Let $b \prec a$ and $f_b : (\prec^{*-1}\{b\}) \to B$ be the function such that

$$\forall y \prec^* b.\ f_b(y) = F(y, f_b \upharpoonright (\prec^{-1}\{y\})).$$

We build a relation $h \subseteq A \times B$ defined by

$$h \stackrel{\text{def}}{=} \bigcup_{b \prec a} f_b$$

Now for any $b \prec a$ there is at least one pair of the form $(b, c) \in h$ for some $c \in B$, because $b \prec^* b$. By the uniqueness property proved before, we have that such a pair is unique. Hence h satisfies the function property. Finally, we let

$$f_a \stackrel{\text{def}}{=} h \cup \{(a, F(a, h))\}$$

to get a function $f_a : (\prec^{*-1}\{a\}) \to B$ such that

$$\forall y \prec^* a.\ f_a(y) = F(y, f_a \upharpoonright (\prec^{-1}\{y\}))$$

proving that $Q(a)$ holds. □

Theorem 4.6 guarantees that, if we (recursively) define f over any $a \in A$ only in terms of the predecessors of a, then f is uniquely determined on all a. Notice that F has a *dependent* type, since the type of its second argument depends on the value of its first argument.

In the following chapters we will exploit fixpoint theory to define the semantics of recursively defined functions. Well-founded recursion gives a simpler method, which however works only in the well-founded case.

Example 4.16 (Product as primitive recursion). Let us consider the Peano formula that defines the product of natural numbers:

$$\begin{aligned} p(0,y) &\stackrel{\text{def}}{=} 0 \\ p(x+1,y) &\stackrel{\text{def}}{=} y + p(x,y) \end{aligned}$$

Let us write the definition in a slightly different way:

$$\begin{aligned} p_y(0) &\stackrel{\text{def}}{=} 0 \\ p_y(x+1) &\stackrel{\text{def}}{=} y + p_y(x) \end{aligned}$$

Let us recast the Peano formula seen above in the formal scheme of well-founded recursion:

$$\begin{aligned} p_y(0) &\stackrel{\text{def}}{=} F_y(0, p_y \restriction \varnothing) = 0 \\ p_y(x+1) &\stackrel{\text{def}}{=} F_y(x+1, p_y \restriction (\prec^{-1}\{x+1\})) = y + p_y(x) \end{aligned}$$

Example 4.17 (Structural recursion). Let us consider the signature Σ for binary trees $A = T_\Sigma$, where $\Sigma_0 = \{0, 1, ...\}$ and $\Sigma_2 = \mathsf{cons}$ (where $\mathsf{cons}(x,y)$ is the constructor for building a tree out of its left and right subtrees). Take the well-founded relation $x_i \prec \mathit{cons}(x_1, x_2)$, $i = 1, 2$. Let $B = \mathbb{N}$.

We want to compute the sum of the elements in the leaves of a binary tree. In Lisp-like notation

$$\mathit{sum}(x) \stackrel{\text{def}}{=} \textbf{if } \mathit{atom}(x) \textbf{ then } x \textbf{ else } \mathit{sum}(\mathit{car}(x)) + \mathit{sum}(\mathit{cdr}(x))$$

where $\mathit{atom}(x)$ returns true if x is a leaf; $\mathit{car}(x)$ denotes the left subtree of x; $\mathit{cdr}(x)$ the right subtree of x. The same function defined in the structural recursion style is

$$\begin{aligned} \mathit{sum}(n) &\stackrel{\text{def}}{=} n \\ \mathit{sum}(\mathsf{cons}(x,y)) &\stackrel{\text{def}}{=} \mathit{sum}(x) + \mathit{sum}(y) \end{aligned}$$

or, according to the well-founded recursive scheme,

$$F(n, sum \restriction \varnothing) \stackrel{\text{def}}{=} n$$
$$F(\mathsf{cons}(x,y), sum \restriction \{x,y\}) \stackrel{\text{def}}{=} sum(x) + sum(y)$$

(where we remind that $\prec^{-1}\{n\} = \varnothing$ and $\prec^{-1}\{\mathsf{cons}(x,y)\} = \{x,y\}$).

For example, for $q \stackrel{\text{def}}{=} \mathsf{cons}(3, \mathsf{cons}(\mathsf{cons}(2,3),4))$ we have

$$\begin{aligned} sum(q) &= sum(3) + sum(\mathsf{cons}(\mathsf{cons}(2,3),4)) \\ &= 3 + (sum(\mathsf{cons}(2,3)) + sum(4)) \\ &= 3 + ((sum(2) + sum(3)) + 4) \\ &= 3 + ((2+3)+4) \\ &= 12 \end{aligned}$$

Example 4.18 (Ackermann function). The Ackermann function is one of the earliest examples of a computable, total recursive function that is not primitive recursive: it grows faster than any such function. The Ackermann function $ack(z,x,y) = ack_y(z,x)$ is defined by well-founded recursion (exploiting the lexicographic precedence relation over pairs of natural numbers) by letting

$$\begin{cases} ack(\ 0\ ,\ 0\ ,\ y\ ) = y \\ ack(\ 0\ ,\ x+1\ ,\ y\ ) = ack(0,x,y)+1 \\ ack(\ 1\ ,\ 0\ ,\ y\ ) = 0 \\ ack(\ z+2\ ,\ 0\ ,\ y\ ) = 1 \\ ack(\ z+1\ ,\ x+1\ ,\ y\ ) = ack(z, ack(z+1,x,y), y) \end{cases}$$

It is immediate to check that from the above definition we have

$$\begin{cases} ack(0,0,y) = y \\ ack(0,x+1,y) = ack(0,x,y)+1 \end{cases} \Rightarrow ack(0,x,y) = y+x$$

$$\begin{cases} ack(1,0,y) = 0 \\ ack(1,x+1,y) = ack(0, ack(1,x,y), y) = ack(1,x,y)+y \end{cases} \Rightarrow ack(1,x,y) = y \times x$$

$$\begin{cases} ack(2,0,y) = 1 \\ ack(2,x+1,y) = ack(1, ack(2,x,y), y) = ack(2,x,y) \times y \end{cases} \Rightarrow ack(2,x,y) = y^x$$

Intuitively: $ack(1,x,y)$ applies addition of y for x times, $ack(2,x,y)$ applies multiplication by y for x times, $ack(3,x,y)$ applies exponentiation to the yth power for x times, and so on. For example, we have

$$\begin{aligned} &ack(0,0,0) = 0+0 = 0 && ack(1,1,1) = 1 \times 1 = 1 \\ &ack(2,2,2) = 2^2 = 4 && ack(3,3,3) = 3^{3^3} = 3^{27} \simeq 7.6 \times 10^{12} \end{aligned}$$

Problems

4.1. Consider the logical system R corresponding to the rules of the grammar

$$S ::= aB \mid bA \qquad A ::= a \mid aS \mid bAA \qquad B ::= b \mid bS \mid aBB$$

where the well-formed formulas are of the form $x \in L_X$, where X is either S or A or B and where x is a string on the alphabet $\{a,b\}$.

1. Write down explicitly the rules in R.
2. Prove by rule induction—in one direction—and by mathematical induction on the length of the strings—in the other direction—that the strings generated by S are all the nonempty strings with the same number of as and bs (i.e., prove the formal predicate $P(x \in L_S) \stackrel{\text{def}}{=} x|_a = x|_b \neq 0$, where $x|_s$ denotes the number of occurrences of the symbol s in the string x), while A generates all the strings with an additional a (formally $P(x \in L_A) \stackrel{\text{def}}{=} x|_a = 1 + x|_b$) and B with an additional b.
3. Finally prove by induction on derivations that

$$P(d/(x \in L_X)) \stackrel{\text{def}}{=} |d| \leq |x|$$

i.e., the depth of any derivation d is less than or equal to the length of the string x generated by it.

4.2. Define by well-founded recursion the function

$$locs : Com \longrightarrow \wp(\mathbf{Loc})$$

that, given a command, returns the set of locations that appear on the left-hand side of some assignment.

Then, prove that $\forall c \in Com, \forall \sigma, \sigma' \in \Sigma$

$$\langle c, \sigma \rangle \to \sigma' \qquad \text{implies} \qquad \forall y \notin locs(c).\ \sigma(y) = \sigma'(y).$$

4.3. Let w denote the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } (x := x - 1\ ;\ y := y + 1).$$

Prove by rule induction that $\forall \sigma, \sigma' \in \Sigma$

$$\langle w, \sigma \rangle \to \sigma' \qquad \text{implies} \qquad \sigma(x) \geq 0 \ \wedge\ \sigma' = \sigma \left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x} \right].$$

4.4. Let R be a binary relation over the set A, i.e., $R \subseteq A \times A$. Let R^+, called the *transitive closure* of R, be the relation defined by the following two rules:

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

1. Prove that for any x and y

$$x\,R^{+}\,y \quad \Leftrightarrow \quad \exists k>0.\ \exists z_0,\ldots,z_k.\ x=z_0 \ \wedge\ z_0\,R\,z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\,R\,z_k \ \wedge\ z_k=y$$

   (*Hint*: Prove the implication $\Rightarrow$ by rule induction and the implication $\Leftarrow$ by induction on the length k of the R-chain.)
2. Give the rules that define instead a relation R' such that

$$x\,R'\,y \quad \Leftrightarrow \quad \exists k\geq 0.\ \exists z_0,\ldots,z_k.\ x=z_0 \ \wedge\ z_0\,R\,z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\,R\,z_k \ \wedge\ z_k=y.$$

4.5. Let IMP^{-} be the language obtained from IMP by removing the **while** construct. Exploit the operational semantics to prove that in IMP^{-}, for every command c the termination property

$$\forall\sigma\in\Sigma.\ \exists\sigma'\in\Sigma.\ \langle c,\sigma\rangle\to\sigma'$$

holds.

4.6. Let us consider the following rules, where m, n and k are positive natural numbers.

$$\frac{}{(m,m)\to m} \qquad \frac{(n,m)\to k}{(m,n)\to k}\ m<n \qquad \frac{(m-n,n)\to k}{(m,n)\to k}\ m>n$$

Prove by rule induction that, for any $n,m,k>0$,

$$(m,n)\to k \qquad \text{implies} \qquad k=\gcd(m,n).$$

where $\gcd(m,n)$ denotes the greatest common divisor of m and n, i.e., if we write $d|j$ to mean that d divides j (in other words, that there exists h such that $j=d\times h$), then $\gcd(n,m)$ is the natural number d such that $d|m\wedge d|n$ and for any d' such that $d'|m\wedge d'|n$ we have $d'\leq d$.
(*Hint*: Prove that any common divisor of m and n with $m>n$ is also a common divisor of $m-n$ and n and vice versa.)

4.7. Prove that, according to the operational semantics of IMP, for any boolean expression b, command c and stores σ, σ'

$$\langle\textbf{while } b \textbf{ do } c,\sigma\rangle\to\sigma' \qquad \text{implies} \qquad \langle b,\sigma'\rangle\to \mathit{false}$$

Explain which induction principle you exploit in the proof.

4.8. Exploit the property from Problem 4.7 to prove that for any $b\in \mathit{Bexp}$ and $c\in \mathit{Com}$ we have $c_1\sim c_2\sim c_3$ where

$$\begin{aligned} c_1 &\stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c\\ c_2 &\stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } (c\,;\textbf{while } b \textbf{ do } c)\\ c_3 &\stackrel{\text{def}}{=} (\textbf{while } b \textbf{ do } c)\,;\ \textbf{while } b \textbf{ do } c \end{aligned}$$

4.9. Define by well-founded recursion the function

$$locs : Aexp \longrightarrow \wp(\mathbf{Loc})$$

that, given an arithmetic expression a, returns the set of locations that occur in a. Use structural induction to show that $\forall a \in Aexp.\ \forall \sigma, \sigma' \in \Sigma.\ \forall n, m \in \mathbb{Z}$

$$\langle a, \sigma \rangle \to n \ \wedge\ \langle a, \sigma' \rangle \to m \ \wedge\ \forall x \in locs(a).\ \sigma(x) = \sigma'(x) \qquad \text{implies} \qquad n = m.$$

4.10. Consider the IMP program

$$w \stackrel{\text{def}}{=} \textbf{while}\ \neg(x = y)\ \textbf{do}\ (x := x + 1\ ;\ y := y - 1)$$

Define the set of stores $T = \{\sigma \mid ...\}$ for which the program w terminates and

1. Prove formally that for any store $\sigma \in T$ there exists σ' such that $\langle w, \sigma \rangle \to \sigma'$. (*Hint*: use well-founded induction on T.)
2. Prove (by using the rule for divergence) that $\forall \sigma \notin T.\ \langle w, \sigma \rangle \not\to$.

4.11. Let us consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while}\ x \neq 0\ \textbf{do}\ x := x - y.$$

1. Prove that, for any σ, σ', if $\langle w, \sigma \rangle \to \sigma'$ then there exists an integer k such that $\sigma(x) = k \times \sigma(y)$.
2. Give a store σ such that $\sigma(x) = k \times \sigma(y)$ for some integer k but such that $\langle w, \sigma \rangle \not\to$. Explain why the program diverges for the given σ.
3. Define a command c such that, for any σ, σ', $\langle c, \sigma \rangle \to \sigma'$ iff $\sigma(x) = k \times \sigma(y)$ for some integer k. Sketch the proof of the double implication.

4.12. Recall that the *height* or *depth* of a derivation d is recursively defined as follows:

$$\begin{aligned} depth(\varnothing / y) &\stackrel{\text{def}}{=} 1 \\ depth(\{d_1, ..., d_n\} / y) &\stackrel{\text{def}}{=} 1 + \max\{depth(d_1), ..., depth(d_n)\} \end{aligned}$$

Given the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while}\ x > 0\ \textbf{do}\ x := x - 1$$

prove by induction on n that for any $\sigma \in \Sigma$ with $\sigma(x) = n \geq 0$ the derivation of

$$\langle w, \sigma \rangle \to \sigma'$$

has depth $n + 3$.

4.13. The *binomial coefficients* $\binom{n}{k}$, with $n, k \in \mathbb{N}$ and $0 \leq k \leq n$, are defined by:

$$\binom{n}{0} \stackrel{\text{def}}{=} 1 \qquad \binom{n}{n} \stackrel{\text{def}}{=} 1 \qquad \binom{n+1}{k+1} \stackrel{\text{def}}{=} \binom{n}{k} + \binom{n}{k+1}$$

Prove that the above definition is given by well-founded recursion, specifying the well-founded relation and the corresponding function $F(b,h)$.

4.14. Consider the well-known sequence of Fibonacci numbers, defined as

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1) + F(n)$$

where $n \in \mathbb{N}$. Explain why the above definition is given by well-founded recursion and make explicit the well-founded relation to be used.

Chapter 5
Partial Orders and Fixpoints

Good old Watson! You are the one fixed point in a changing age.
(Sherlock Holmes)

Abstract This chapter is devoted to the introduction of the foundations of the denotational semantics of computer languages. The concepts of complete partial orders with bottom and of monotone and continuous functions are introduced and then the main fixpoint theorem is presented. The chapter is concluded by studying the immediate consequence operator, which is used to relate logical systems and fixpoint theory.

5.1 Orders and Continuous Functions

As we have seen, the operational semantics gives us a very concrete semantics, since the inference rules describe step by step the bare essential operations on the state required to reach the final state of computation. Unlike the operational semantics, the denotational one provides a more abstract view. Indeed, the denotational semantics gives us directly the meaning of the constructs of the language as particular functions over domains. Domains are sets whose structure will ensure the correctness of the constructions of the semantics.

As we will see, one of the most attractive features of the denotational semantics is that it is compositional, namely, *the meaning of a composite program is given by combining the meanings of its constituents*. The compositional property of denotational semantics is obtained by defining the semantics by structural recursion. Obviously there are particular issues in defining the interpretation of the "while" construct of IMP, since the semantics of this construct, as we saw in the previous chapters, is inherently recursive. General recursion is forbidden in structural recursion, which allows only the use of subterms. The solution to this problem is given by solving equations of the type $x = f(x)$, namely by finding the fixpoint(s) of suitable functions f. On the one hand we would like to ensure that each recursive definition that we introduce has a fixpoint. Therefore we will restrict our study to a particular class of functions: continuous functions. On the other hand, the aim of the theory we will develop, called domain theory, will be to identify one solution when more than one is

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_5

available, and to provide an approximation method for computing it, which is given by the fixpoint theorem (Theorem 5.6).

5.1.1 Orders

We introduce the general theory of partial orders, which will bring us to the concept of domain.

Definition 5.1 (Partial order). A *partial order* is a pair $(P, \sqsubseteq_P)$ where P is a set and $\sqsubseteq_P \subseteq P \times P$ is a binary relation (i.e., it is a set of pairs of elements of P) which is

reflexive:	$\forall p \in P.\ p \sqsubseteq_P p$
antisymmetric:	$\forall p, q \in P.\ p \sqsubseteq_P q \wedge q \sqsubseteq_P p \Longrightarrow p = q$
transitive:	$\forall p, q, r \in P.\ p \sqsubseteq_P q \wedge q \sqsubseteq_P r \Longrightarrow p \sqsubseteq_P r$

We call $(P, \sqsubseteq_P)$ a *poset* (for *partially ordered set*).

We will conveniently omit the subscript P from $\sqsubseteq_P$ when no confusion can arise. We write $p \sqsubset q$ when $p \sqsubseteq q$ and $p \neq q$.

Example 5.1 (Powerset). Let $(\wp(S), \subseteq)$ be the powerset of a set S together with the inclusion relation. It is easy to see that $(\wp(S), \subseteq)$ is a poset.

reflexive:	$\forall s \subseteq S.\ s \subseteq s$
antisymmetric:	$\forall s_1, s_2 \subseteq S.\ s_1 \subseteq s_2 \subseteq s_1 \Rightarrow s_1 = s_2$
transitive:	$\forall s_1, s_2, s_3 \subseteq S.\ s_1 \subseteq s_2 \subseteq s_3 \Rightarrow s_1 \subseteq s_3$

Actually, partial orders are a generalisation of the concept of powerset ordered by inclusion. Thus we should not be surprised by this result.

Remark 5.1 (Partial orders vs well-founded relations). Partial order relations should not be confused with the well-founded relations studied in the previous chapter. In fact

- Any well-founded relation (on a nonempty set) is not reflexive (otherwise an infinite descending chain could be constructed by iterating over the same element).
- Any well-founded relation is antisymmetric (the premise $p \sqsubseteq q \wedge q \sqsubseteq p$ must be always false, otherwise an infinite descending chain could be constructed).
- A well-founded relation can be transitive, but it is not necessarily so (e.g., the immediate precedence relation over natural numbers is well-founded but not transitive; instead the 'less than' relation is well-founded and transitive).
- Any (nonempty) partial order has an infinite descending chain (take any element p and the chain $p \sqsupseteq p \sqsupseteq p \ldots$) and is thus non-well-founded.
- If we take the relation $\sqsubset$ induced by a partial order $\sqsubseteq$, then it can be well-founded, but it is not necessarily so (e.g., the strict inclusion relation over $\wp(\mathbb{N})$ has an infinite descending chain whose ith element is the set $\{n \mid n \in \mathbb{N} \wedge n \geq i\}$).

- If we take the reflexive and transitive closure $\prec^*$ of a well-founded relation $\prec$, then it is a partial order (reflexivity and transitivity are obvious; for the antisymmetric property, suppose that there were two elements $p \neq q$ such that $p \prec^* q \wedge q \prec^* p$ then there would be a cycle over p using $\prec$, contradicting the assumption that $\prec$ is well-founded).

Two elements $p, q \in P$ are called *comparable* if $p \sqsubseteq q$ or $q \sqsubseteq p$. When any two elements of a partial order are comparable, then it is called a *total* order.

Definition 5.2 (Total order). Let $(P, \sqsubseteq)$ be a partial order such that

$$\forall p, q \in P.\ p \sqsubseteq q \ \vee\ q \sqsubseteq p$$

We call $(P, \sqsubseteq)$ a *total order*.

Example 5.2. Given a set S, its powerset $(\wp(S), \subseteq)$ ordered by inclusion is a total order if and only if $|S| \leq 1$. In fact, in one direction suppose that $(\wp(S), \subseteq)$ is a total order and take $p, q \in S$; clearly $\{p\} \subseteq \{q\} \vee \{q\} \subseteq \{p\}$ holds only when $p = q$, i.e., S must have at most one element. Vice versa, if $S = \varnothing$ then $\wp(S) = \{\varnothing\}$ and $\varnothing \subseteq \varnothing$; if $S = \{p\}$ for some p, then $\wp(S) = \{\varnothing, \{p\}\}$ and $\varnothing \subseteq \varnothing \subseteq \{p\} \subseteq \{p\}$.

Theorem 5.1 (Subsets of an order). *Let $(P, \sqsubseteq_P)$ be a partial order and let $Q \subseteq P$. Then $(Q, \sqsubseteq_Q)$ is a partial order, with $\sqsubseteq_Q \stackrel{\text{def}}{=} \sqsubseteq_P \cap (Q \times Q)$. Similarly, if $(P, \sqsubseteq_P)$ is a total order then $(Q, \sqsubseteq_Q)$ is a total order.*

The proof is left as an easy exercise for the reader (see Problem 5.1).

Let us see some examples that will be very useful in understanding the concepts of partial and total orders.

Example 5.3 (Natural Numbers). Let $(\mathbb{N}, \leq)$ be the set of natural numbers with the usual order; $(\mathbb{N}, \leq)$ is a total order.

reflexive:	$\forall n \in \mathbb{N}.\ n \leq n$
antisymmetric:	$\forall n, m \in \mathbb{N}.\ n \leq m \wedge m \leq n \Longrightarrow m = n$
transitive:	$\forall n, m, z \in \mathbb{N}.\ n \leq m \wedge m \leq z \Longrightarrow n \leq z$
total:	$\forall n, m \in \mathbb{N}.\ n \leq m \vee m \leq n$

Example 5.4 (Discrete order). Let $(P, \sqsubseteq)$ be a partial order defined as follows:

$$\forall p \in P.\ p \sqsubseteq p$$

We call $(P, \sqsubseteq)$ a *discrete order*.

Example 5.5 (Flat order). A *flat order* is a partial order $(P, \sqsubseteq)$ for which there exists an element $\bot \in P$ such that

$$\forall p, q \in P.\ p \sqsubseteq q \Leftrightarrow\ p = \bot \vee p = q$$

The element $\bot$ is called *bottom* and it is unique. In fact, suppose that two such elements $\bot_1, \bot_2$ exist. Then, we have $\bot_1 \sqsubseteq \bot_2$ and also $\bot_2 \sqsubseteq \bot_1$; thus by antisymmetry we have $\bot_1 = \bot_2$.

5.1.2 Hasse Diagrams

The aim of this section is to provide a tool that allows us to represent orders in a comfortable way.

First of all we might think to use graphs to represent an order. In this framework each element of the order is represented by a node of the graph and the order relation by the arrows (i.e., we have an arrow from a to b if and only if $a \sqsubseteq b$).

This notation is not very manageable, indeed we repeat many times redundant information. For example in the usual order on natural numbers we would have $n+1$ incoming arrows and infinitely many outgoing arrows for a node labelled by n. We need a more compact notation, which leaves implicit some information that can be inferred by exploiting the properties of partial orders. This notation is represented by Hasse diagrams. The idea is to omit 1) every reflexive arc (from a node to itself), because we know by reflexivity that such an arc is present for every node; and 2) every arc from a to c when there is a node b with an arc from a to b and one from b to c, because the presence of the arc from a to c can be inferred by transitivity.

Definition 5.3 (Hasse diagram). Given a poset $(P, \sqsubseteq)$, let R be the binary relation defined by:

$$\frac{x \sqsubseteq y \quad y \sqsubseteq z \quad x \neq y \neq z}{xRz} \qquad \frac{\varnothing}{xRx}$$

We call a *Hasse diagram* the relation H defined as

$$H \stackrel{\text{def}}{=} \sqsubseteq \setminus R$$

Note that the first rule can be written more concisely as

$$\frac{x \sqsubset y \quad y \sqsubset z}{xRz}$$

The Hasse diagram omits the information deducible by transitivity and reflexivity. A simple example of a Hasse diagram is in Figure 5.1.

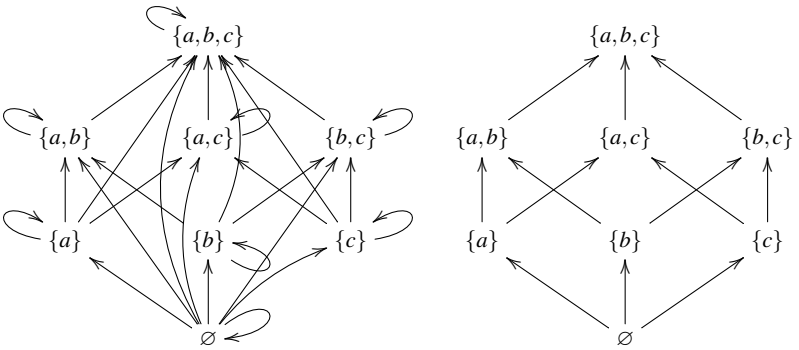
To ensure that all the needed information is contained in the Hasse diagram we rely on the following theorem.

Theorem 5.2 (Order relation, Hasse diagram equivalence). *Let $(P, \sqsubseteq)$ be a partial order with P a finite set, and let H be its Hasse diagram. Then, the transitive and reflexive closure H^* of H is equal to $\sqsubseteq$.*

Proof. Formally, we want to prove the two inclusions $H^* \subseteq \sqsubseteq$ and $\sqsubseteq \subseteq H^*$ separately, where the relation H^* is defined by the inference rules below:

$$\frac{}{x H^* x} \qquad \frac{x H y}{x H^* y} \qquad \frac{x H^* y \wedge y H^* z}{x H^* z}$$

$H^* \subseteq \sqsubseteq$: Suppose $x\, H^*\, y$. Then, there exist (see Problem 4.4) $k \in \mathbb{N}$ and $z_0, \ldots, z_k$ such that



(a) $\wp(\{a,b,c\})$ ordered by inclusion (b) Hasse diagram for $\wp(\{a,b,c\})$

Fig. 5.1: Hasse diagram for the powerset over $\{a,b,c\}$ ordered by inclusion

$$x = z_0 \ \wedge \ z_0 \, H \, z_1 \ \wedge \ \ldots \wedge \ z_{k-1} \, H \, z_k \ \wedge \ z_k = y$$

Since $H \ \subseteq \ \sqsubseteq$ by definition, we have

$$x = z_0 \ \wedge \ z_0 \sqsubseteq z_1 \ \wedge \ \ldots \wedge \ z_{k-1} \sqsubseteq z_k \ \wedge \ z_k = y$$

Hence, by transitivity of $\sqsubseteq$ it follows that $x \sqsubseteq y$.

$\sqsubseteq \ \subseteq \ H^*$: Given $x \sqsubseteq y$, let us denote by $]x,y[$ the set of elements strictly contained between x and y, i.e.,

$$]x,y[\stackrel{\text{def}}{=} \{z \mid x \sqsubset z \ \wedge \ z \sqsubset y\}.$$

Clearly $]x,y[$ is finite because P is finite. We prove that $x H^* y$ by mathematical induction on the number of elements in $]x,y[$.

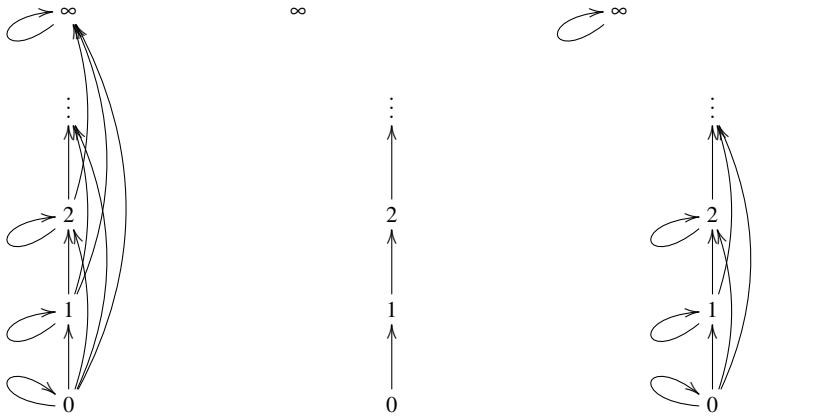
Base case: When $]x,y[$ is empty, it means that $(x,y) \notin R$. Hence $x H y$ and thus $x H^* y$.

Inductive case: Suppose $]x,y[$ has $n+1$ elements. Take $z \in]x,y[$. Clearly the sizes of $]x,z[$ and $]z,y[$ are strictly smaller than that of $]x,y[$, and since $x \sqsubset z$ and $z \sqsubset y$, by the inductive hypothesis it follows that $x H^* z$ and $z H^* y$. Hence $x H^* y$. □

The above theorem only allows us to represent finite orders.

Example 5.6 (Infinite order). Let us see that Hasse diagrams do not work well with infinite orders. Let $\Omega = (\mathbb{N} \cup \{\infty\}, \leq)$ be the usual order on natural numbers extended with a top element ∞ such that $n \leq \infty$ and $\infty \leq \infty$, i.e.,

$$0 \leq 1 \leq 2 \leq \cdots \leq n \leq \cdots \leq \infty$$



(a) $(\mathbb{N}\cup\{\infty\},\leq)$ (b) H induced by $(\mathbb{N}\cup\{\infty\},\leq)$ (c) H^* induced by $(\mathbb{N}\cup\{\infty\},\leq)$

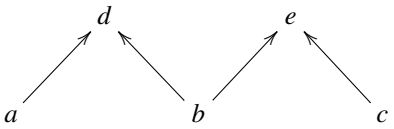
Fig. 5.2: Infinite orders and Hasse diagrams

From Definition 5.3 it follows that for any $n\in\mathbb{N}$ we have $n\,R\,\infty$ (because $n<n+1<\infty$) and that for any $n,k\in\mathbb{N}$ it holds that $n\,R\,n+2+k$ (because $n<n+1<n+2+k$). Moreover, for any $x\in\mathbb{N}\cup\{\infty\}$ we have $x\,R\,x$. In particular, the Hasse diagram eliminates the arc between each natural number and ∞. Now using the transitive and reflexive closure we would like to get back the original order. Using the inference rules we obtain the usual order on natural numbers without any relation between ∞ and the natural numbers (recall that we only allow finite proofs). The situation is illustrated in Figure 5.2

Definition 5.4 (Least element). Let $(P,\sqsubseteq_P)$ be a partial order and take $Q\subseteq P$. An element $\ell\in Q$ is a *least* element of $(Q,\sqsubseteq_Q)$ if

$$\forall q\in Q.\ \ell\sqsubseteq_Q q$$

Example 5.7 (No least element). Let us consider the order associated with the Hasse diagram



The sets $\{a,b,d\}$ and $\{a,b,c,d,e\}$ have no least element. As we will see, the elements a,b and c are minimal since they have no smaller elements in the order.

Theorem 5.3 (Uniqueness of the least element). *Let $(P,\sqsubseteq)$ be a partial order. P has at most one least element.*

Proof. Let $\ell_1, \ell_2 \in P$ both be least elements of P, then $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$. Now by using the antisymmetric property we get $\ell_1 = \ell_2$. □

The counterpart of the least element is the concept of *greatest* element. We can define the greatest element as the least element of the reverse order $\sqsubseteq^{-1}$ (defined by letting $x \sqsubseteq^{-1} y \Leftrightarrow y \sqsubseteq x$).

Definition 5.5 (Minimal element). Let $(P, \sqsubseteq_P)$ be a partial order and take $Q \subseteq P$. An element $m \in Q$ is a *minimal element* of $(Q, \sqsubseteq_Q)$ if

$$\forall q \in Q.\ q \sqsubseteq_Q m \Rightarrow q = m$$

As for the least element we have the dual of minimal elements, called *maximal elements*: They are the minimal elements of the reverse order $\sqsubseteq^{-1}$.

Remark 5.2 (Least vs minimal elements). Note that the definition of minimal and least element (maximal and greatest) are quite different:

- The least element ℓ is the (unique) smallest element of a set.
- A minimal element m is just such that no smaller element can be found in the set, i.e., $\forall q \in Q. q \not\sqsubset m$ (but there is no guarantee that all the elements $q \in Q$ are comparable with m).
- The least element of an order is obviously minimal, but a minimal element is not necessarily the least.

Definition 5.6 (Upper bound). Let $(P, \sqsubseteq)$ be a partial order and $Q \subseteq P$ be a subset of P, then $u \in P$ is an *upper bound* of Q if

$$\forall q \in Q.\ q \sqsubseteq u$$

Note that unlike a maximal element and the greatest element an upper bound does not necessarily belong to the subset Q of elements we are considering.

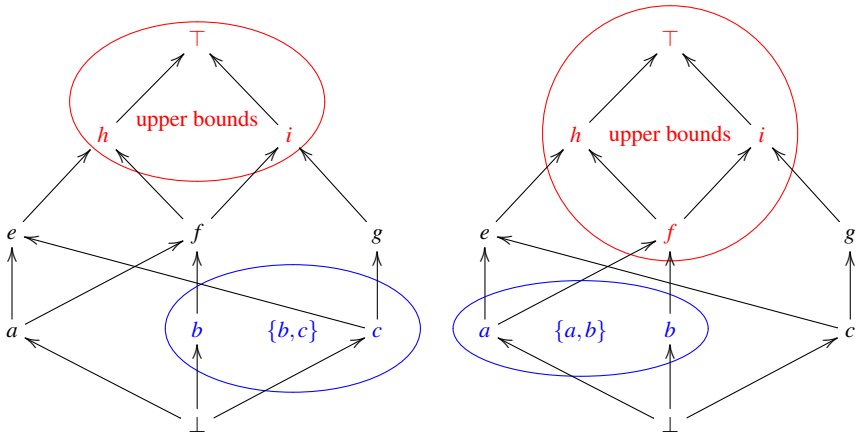
Definition 5.7 (Least upper bound). Let $(P, \sqsubseteq)$ be a partial order and $Q \subseteq P$ be a subset of P. Then, $p \in P$ is the *least upper bound* (*lub*) of Q if and only if p is the least element of the upper bounds of Q. Formally, we require that

1. p is an upper bound of Q ($\forall q \in Q.\ q \sqsubseteq p$);
2. for any upper bound u of Q, then $p \sqsubseteq u$ ($\forall u \in P.\ (\forall q \in Q.\ q \sqsubseteq u) \Rightarrow p \sqsubseteq u$);

and we write $lub(Q) = p$.

It follows immediately from Theorem 5.3 that the least upper bound, when it exists, is unique.

Example 5.8 (lub). Now we will clarify the concept of lub with two examples. Let us consider the order represented by the Hasse diagram in Figure 5.3 (a). The set of upper bounds of the subset $\{b, c\}$ is the set $\{h, i, \top\}$. This set has no least element (i.e., h and i are not comparable) so the set $\{b, c\}$ has no lub. In Figure 5.3 (b) we see that the set of upper bounds of the set $\{a, b\}$ is the set $\{f, h, i, \top\}$. The least element of the latter set is f, which is thus the lub of $\{a, b\}$.



(a) The subset $\{b,c\}$ has no lub

(b) The subset $\{a,b\}$ has lub f

Fig. 5.3: Two subsets of a poset, and their upper bounds

5.1.3 Chains

One of the main concepts in the study of partial orders is that of a *chain*, which is formed by taking a subset of totally ordered elements.

Definition 5.8 (Chain). Let $(P,\sqsubseteq)$ be a partial order, we call a *chain* a function $C : \mathbb{N} \to P$ such that

$$\forall n \in \mathbb{N}.\ C(n) \sqsubseteq C(n+1)$$

We will often write $C = \{d_i\}_{i\in\mathbb{N}}$, where $\forall i \in \mathbb{N}.\ d_i = C(i)$, i.e.,

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$$

Definition 5.9 (Finite chain). Let $C : \mathbb{N} \to P$ be a chain such that the image of C is a finite set, then we say that C is a *finite chain*. Otherwise we say that C is infinite.

Note that a finite chain still has infinitely many elements $\{d_i\}_{i\in\mathbb{N}}$, but only finitely many different ones. In particular, it has one index k and one element d such that $\forall i \in \mathbb{N}.\ d_{k+i} = d$.

Example 5.9 (Finite and infinite chains). Take the partial order $(\mathbb{N},\leq)$. The chain of even numbers

$$0 \leq 2 \leq 4 \leq \cdots$$

is an infinite chain. Instead, the constant chain

$$1 \leq 1 \leq 1 \leq \cdots$$

is a finite chain.

Definition 5.10 (Limit of a chain). Let C be a chain. The lub of the image of C, if it exists, is called *the limit of* C. If d is the limit of the chain $C = \{d_i\}_{i\in\mathbb{N}}$, we write $d = \bigsqcup_{i\in\mathbb{N}} d_i$.

Remark 5.3. Each finite chain has a limit. Indeed each finite chain has a finite totally ordered image: obviously this set has a lub (the greatest element of the set).

Lemma 5.1 (Prefix independence of the limit). *Let $n \in \mathbb{N}$ and let C and C' be two chains such that $C = \{d_i\}_{i\in\mathbb{N}}$ and $C' = \{d_{n+i}\}_{i\in\mathbb{N}}$. Then C and C' have the same limit, if any.*

Proof. Let us prove a stronger property, namely that the chains C and C' have the same set of upper bounds.

Obviously if c is an upper bound of C, then c is an upper bound of C', since each element of C' is contained in C.

Vice versa if c is an upper bound of C', we need to show that $\forall j \in \mathbb{N}.\ j \leq n \Rightarrow d_j \sqsubseteq c$. Since $d_n \sqsubseteq c$ and $\forall j \in \mathbb{N}.\ j \leq n \Rightarrow d_j \sqsubseteq d_n$ by transitivity of $\sqsubseteq$ it follows that c is an upper bound of C.

Now since C and C' have the same set of upper bound elements, they have the same lub, if it exists at all. □

The main consequence of Lemma 5.1 is that we can always eliminate from or add a finite prefix to a chain preserving the limit.

A stronger result guarantees that any infinite subsequence of a chain C has the same set of upper bounds as C and thus the same limit, if any (see Problem 5.13).

5.1.4 Complete Partial Orders

The aim of partial orders and continuous functions is to provide a framework that allows the definition of the denotational semantics when recursive equations are needed. Complete partial orders extend the concept of partial orders to support the limit operation on chains, which is a generalisation of the countable union operation on a powerset. Limits will have a key role in finding fixpoint solutions to recursive equations.

Definition 5.11 (Complete partial orders). Let $(P, \sqsubseteq)$ be a partial order. We say that $(P, \sqsubseteq)$ is *complete* (CPO) if each chain has a limit (i.e., each chain has a lub).

From Remark 5.3, it follows immediately that if a partial order has only finite chains then it is complete.

Definition 5.12 (CPO with bottom). Let $(D, \sqsubseteq)$ be a CPO, we say that $(D, \sqsubseteq)$ is a *CPO with bottom* ($CPO_\perp$) if it has a least element $\perp$ (called *bottom*).

Let us see some examples that will clarify the concept of CPO. To avoid ambiguities, sometimes we will denote the bottom element of the CPO D by $\perp_D$.

Example 5.10 (Powerset completeness). Let us consider again the previous example of powerset (Example 5.1). We show that the partial order $(\wp(S), \subseteq)$ is complete. Take any chain $\{S_i\}_{i\in\mathbb{N}}$ of subsets of S. Then:

$$lub(S_0 \subseteq S_1 \subseteq S_2 \ldots) = \{d \mid \exists k \in \mathbb{N}.\, d \in S_k\} = \bigcup_{i\in\mathbb{N}} S_i \in \wp(S)$$

Example 5.11 (Partial order without upper bounds). Now let us take the usual order on natural numbers $(\mathbb{N}, \leq)$. Obviously all its finite chains have a limit (i.e., the greatest element of the chain). Vice versa infinite chains have no limits (i.e., there is no natural number greater than infinitely many natural numbers). To make the order a CPO all we have to do is to add an element greater than all the natural numbers. So we add the element ∞ and extend the order relation by letting $x \leq \infty$ for all $x \in \mathbb{N} \cup \{\infty\}$. The new poset $(\mathbb{N} \cup \{\infty\}, \leq)$ is a CPO, because ∞ is the limit of any infinite chain.

Remark 5.4 (A subset of a CPO is not necessarily a CPO). We have seen that when we restrict an order relation to a subset of elements we still get a PO (see Theorem 5.1). The previous example shows that, in general, the same property does not hold at the level of CPOs. The problem is due to the fact that, given a chain whose elements are all in the subset, the lub of the chain is not necessarily in the subset.

Example 5.12 (Partial order without least upper bound). Let us define the partial order $(\mathbb{N} \cup \{\infty_1, \infty_2\}, \sqsubseteq)$ as follows:

$$(\sqsubseteq \restriction \mathbb{N}) = \leq, \quad \forall x \in \mathbb{N} \cup \{\infty_1\}.\, x \sqsubseteq \infty_1, \quad \forall x \in \mathbb{N} \cup \{\infty_2\}.\, x \sqsubseteq \infty_2$$

where $\sqsubseteq \restriction \mathbb{N}$ is the restriction of $\sqsubseteq$ to natural numbers. This partial order is not complete, indeed each infinite chain has two upper bounds (i.e., ∞_1 and ∞_2) which are not comparable, hence there is no least upper bound.

The next example illustrates a fundamental CPO that will be exploited in the next chapters: the set of partial functions on natural numbers.

Example 5.13 (Partial functions). Let $\mathbf{Pf} \stackrel{\text{def}}{=} (\mathbb{N} \rightharpoonup \mathbb{N})$ be the set of partial functions from natural numbers to natural numbers. Recall that a partial function is a relation $f \subseteq \mathbb{N} \times \mathbb{N}$ with the *functional* property:

$$\forall n, m, k \in \mathbb{N}.\, n\, f\, m \wedge n\, f\, k \;\Rightarrow\; m = k$$

So the set **Pf** can be viewed as:

$$\mathbf{Pf} \stackrel{\text{def}}{=} \{f \subseteq \mathbb{N} \times \mathbb{N} \mid \forall n, m, k \in \mathbb{N}.\, n\, f\, m \wedge n\, f\, k \Rightarrow m = k\}$$

Let us denote by $f(n)\downarrow$ the predicate $\exists m \in \mathbb{N}.\, (n, m) \in f$ (i.e., $f(n)\downarrow$ holds when the function f is defined on n). Now it is easy to define a partial order $\sqsubseteq$ on **Pf**. We let

$$f \sqsubseteq g \;\Leftrightarrow\; (\forall n \in \mathbb{N}.\, f(n)\downarrow \Rightarrow\; (g(n)\downarrow \wedge f(n) = g(n)))$$

Thus f precedes g if whenever f is defined on n also g is defined on n and $f(n) = g(n)$. When $f(n)$ is not defined, then $g(n)$ can be defined and take any value. When both f and g are seen as (functional) relations, then the above definition boils down to checking that f is included in g. Of course, the poset $(\wp(\mathbb{N} \times \mathbb{N}), \subseteq)$ has the empty relation as bottom element (i.e., the function undefined everywhere), and each infinite chain has as limit the countable union of the relations in the chain.

To show that **Pf** is complete, we need to show that the limits of chains whose elements are in **Pf** also satisfy the functional property, i.e., they are elements of **Pf**.

Theorem 5.4. *Let $f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$ be a chain in* **Pf***, i.e., each relation f_i satisfies the functional property, i.e.,*

$$\forall i \in \mathbb{N}.\ \forall n,m,k \in \mathbb{N}.\ n\, f_i\, m \wedge n\, f_i\, k \Rightarrow m = k$$

Then, the relation $f \stackrel{\text{def}}{=} \cup_{i\in\mathbb{N}} f_i$ satisfies the functional property, namely:

$$\forall n,m,k \in \mathbb{N}.\ n\, f\, m \wedge n\, f\, k \Rightarrow m = k$$

Proof. Let us take generic $n,m,k \in \mathbb{N}$ such that the premise $n\, f\, m \wedge n\, f\, k$ of the implication holds. We need to prove the consequence $m = k$. By $n\, f\, m$, there exists $j \in \mathbb{N}$ with $n\, f_j\, m$ and, by $n\, f\, k$ there exists $h \in \mathbb{N}$ with $n\, f_h\, k$. We take $o = \max\{j,h\}$, then it holds that $n\, f_o\, m \wedge n\, f_o\, k$. Since $f_o \in$ **Pf**, it satisfies the functional property and thus from $n\, f_o\, m \wedge n\, f_o\, k$ we can conclude that $m = k$. □

Example 5.14 (Partial functions as total functions). Let us show a second way to define a CPO on the partial functions on natural numbers. Let $\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ and $(\mathbb{N}_\perp, \sqsubseteq_{\mathbb{N}_\perp})$ be the flat order obtained by adding $\perp$ to the discrete order of the natural numbers. In other words we have $x \sqsubseteq_{\mathbb{N}_\perp} y$ iff $x = y$ or $x = \perp$. Then take the set of total functions $\mathbf{Tf} = (\mathbb{N} \to \mathbb{N}_\perp)$. Equivalently

$$\mathbf{Tf} \stackrel{\text{def}}{=} \{ f \subseteq \mathbb{N} \times \mathbb{N}_\perp \mid (\forall n,m,k \in \mathbb{N}.\ n\, f\, m \wedge n\, f\, k \Rightarrow m = k) \wedge (\forall n \in \mathbb{N}.\ \exists x \in \mathbb{N}_\perp.\ n\, f\, x) \}$$

We define the following order on **Tf**

$$f \sqsubseteq g \ \Leftrightarrow\ \forall n \in \mathbb{N}.\ f(n) \sqsubseteq_{\mathbb{N}_\perp} g(n).$$

That is, if $f(n) = \perp$ then $g(n)$ can assume any value, including $\perp$; otherwise it must be that $g(n) = f(n)$. The bottom element of the order is the function that returns $\perp$ for every argument. Note that the above order is complete. In fact, the limit of a chain obviously exists as a relation, and it is easy to show, analogously to the partial function case, that it is in addition a total function. The proof is left as an exercise for the reader (see Problem 5.11).

Example 5.15 (Limit of a chain of partial functions). Let $\{f_i : \mathbb{N} \to \mathbb{N}_\perp\}_{i\in\mathbb{N}}$ be a chain in **Tf** such that for any $i \in \mathbb{N}$ we have

$$f_i(n) \stackrel{\text{def}}{=} \begin{cases} 3 & \text{if } n \leq i \wedge 2 \mid n \\ \bot & \text{otherwise} \end{cases}$$

where the predicate $k \mid n$ is true when k divides n (i.e., $2 \mid n$ is true when n is even and false otherwise). Let us consider some evaluations of the functions f_i with $i \in [0,4]$:

$$\begin{array}{lllll} f_0(0)=3 & f_0(1)=\bot & f_0(2)=\bot & f_0(3)=\bot & f_0(4)=\bot \cdots \\ f_1(0)=3 & f_1(1)=\bot & f_1(2)=\bot & f_1(3)=\bot & f_1(4)=\bot \cdots \\ f_2(0)=3 & f_2(1)=\bot & f_2(2)=3 & f_2(3)=\bot & f_2(4)=\bot \cdots \\ f_3(0)=3 & f_3(1)=\bot & f_3(2)=3 & f_3(3)=\bot & f_3(4)=\bot \cdots \\ f_4(0)=3 & f_4(1)=\bot & f_4(2)=3 & f_4(3)=\bot & f_4(4)=3 \ \cdots \end{array}$$

Thus the limit of the chain is the function f that returns 3 when applied to even numbers and $\bot$ otherwise:

$$f(n) \stackrel{\text{def}}{=} \begin{cases} 3 & \text{if } 2 \mid n \\ \bot & \text{otherwise} \end{cases}$$

In general, the limit $f \stackrel{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} f_i$ of a chain in **Tf** is a function $f : \mathbb{N} \to \mathbb{N}_\bot$ such that $f(n) = m$ for some $m \neq \bot$ if and only if there exists an index $k \in \mathbb{N}$ with $f_k(n) = m$. Note also that when $i \leq j$ and $f_i(n) \neq \bot$ it must be the case that $f_j(n) = f_i(n)$. On the contrary, when $i \leq j$ and $f_j(n) = \bot$ it means that $f_i(n) = \bot$.

5.2 Continuity and Fixpoints

5.2.1 Monotone and Continuous Functions

In order to define a class of functions over CPOs which ensures the existence of their fixpoints we introduce two general properties of functions: *monotonicity* and *continuity*.

Definition 5.13 (Monotonicity). Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be two CPOs. We say that a function $f : D \to E$ is *monotone* if

$$\forall d, d' \in D.\ d \sqsubseteq_D d' \Rightarrow f(d) \sqsubseteq_E f(d')$$

We say that a monotone function *preserves the order*. So if $\{d_i\}_{i \in \mathbb{N}}$ is a chain in $(D, \sqsubseteq_D)$ and $f : D \to E$ is a monotone function, then $\{f(d_i)\}_{i \in \mathbb{N}}$ is a chain in $(E, \sqsubseteq_E)$. Often we will consider functions whose domain and codomain coincide (i.e., $E = D$), in which case we just say that f is a function on $(D, \sqsubseteq_D)$.

Example 5.16 (Non-monotone function). Let us define a CPO $(\{\bot, 0, 1\}, \sqsubseteq)$ such that $\bot \sqsubseteq 0$, $\bot \sqsubseteq 1$ and $x \sqsubseteq x$ for any $x \in \{\bot, 0, 1\}$. Now define a function f on $(\{\bot, 0, 1\}, \sqsubseteq)$ as follows:

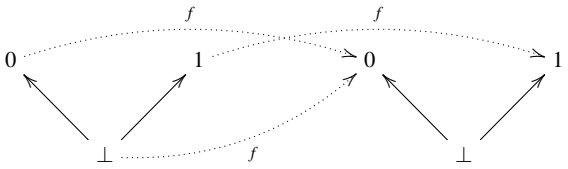


Fig. 5.4: A non-monotone function

$$f(\bot) = 0 \quad f(0) = 0 \quad f(1) = 1$$

This function is not monotone, indeed $\bot \sqsubseteq 1$ but $f(\bot) = 0$ and $f(1) = 1$ are not comparable (see Figure 5.4), so the function f does not preserve the order.

Continuity guarantees that taking the image of the limit of a chain is the same as taking the limit of the images of the elements in the chain.

Definition 5.14 (Continuity). Let $(D, \sqsubseteq_D)$ and $(E \sqsubseteq_E)$ be two CPOs and let $f : D \to E$ be a monotone function. We say that f is a continuous function if for each chain in $(D, \sqsubseteq)$ we have

$$f(\bigsqcup_{i \in \mathbb{N}} d_i) = \bigsqcup_{i \in \mathbb{N}} f(d_i)$$

As is the case for most definitions of continuity, the operations of applying a function and taking the limit can be exchanged. For this reason, we say that a continuous function *preserves limits*. Moreover, note that the limit $\bigsqcup_{i \in \mathbb{N}} d_i$ is taken in D, while the limit $\bigsqcup_{i \in \mathbb{N}} f(d_i)$ is taken in E.

Remark 5.5. Let $(D, \sqsubseteq)$ be a CPO that has only finite chains. Then any chain $\{d_i\}_{i \in \mathbb{N}}$ in D is such that there are $d \in D$ and $k \in \mathbb{N}$ such that $\forall i \in \mathbb{N}.\ d_{i+k} = d$ and it has a limit (d) that is also an element of the chain. Thus any monotone function $f : D \to E$ is continuous, because $\forall i \in \mathbb{N}.\ f(d_{i+k}) = f(d)$ (i.e., the chain $\{f(d_i)\}_{i \in \mathbb{N}}$ is finite and its limit is $f(d)$).

Interestingly, continuous functions are closed under composition.

Theorem 5.5 (Continuity of composition). *Let $(D, \sqsubseteq_D)$, $(E \sqsubseteq_E)$ and $(F \sqsubseteq_F)$ be three CPOs, and $f : D \to E$, $g : E \to F$ be two continuous functions. Their composition*

$$h \stackrel{\text{def}}{=} g \circ f : D \to F$$

defined by letting $h(d) = g(f(d))$ for all $d \in D$ is continuous.

Proof. Let $\{d_i\}_{i \in \mathbb{N}}$ be a chain in D. We want to prove that $h(\bigsqcup_{i \in \mathbb{N}} d_i) = \bigsqcup_{i \in \mathbb{N}} h(d_i)$. We have

$$\begin{aligned} h(\bigsqcup_{i\in\mathbb{N}} d_i) &= g(f(\bigsqcup_{i\in\mathbb{N}} d_i)) && \text{by definition of } h = g\circ f \\ &= g(\bigsqcup_{i\in\mathbb{N}} f(d_i)) && \text{by continuity of } f \\ &= \bigsqcup_{i\in\mathbb{N}} g(f(d_i)) && \text{by continuity of } g \\ &= \bigsqcup_{i\in\mathbb{N}} h(d_i) && \text{by definition of } h = g\circ f \end{aligned}$$

□

Remark 5.6. The composition $g \circ f$ is sometimes denoted also by $f;g$.

Example 5.17 (A monotone function which is not continuous). Let $(\mathbb{N}\cup\{\infty\},\leq)$ be the CPO from Example 5.11. Define a function $f : \mathbb{N}\cup\{\infty\} \to \mathbb{N}\cup\{\infty\}$ such that:

$$f(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \in \mathbb{N} \\ 1 & \text{if } x = \infty \end{cases}$$

It is immediate to check that f is monotone:

- for $n,m \in \mathbb{N}$, if $n \leq m$ we have $f(n) = 0 \leq 0 = f(m)$;
- for $n \in \mathbb{N}$, we have $n \leq \infty$ and $f(n) = 0 \leq 1 = f(\infty)$;
- for $\infty \leq \infty$ we have of course $f(\infty) \leq f(\infty)$.

Let us consider the chain $\{d_i\}_{i\in\mathbb{N}}$ of even numbers:

$$0 \leq 2 \leq 4 \leq 6 \leq \ldots$$

whose limit is ∞. The chain $\{f(d_i)\}_{i\in\mathbb{N}}$ is instead the constant chain

$$0 \leq 0 \leq 0 \leq 0 \leq \ldots$$

whose limit is 0. So we have

$$f(\bigsqcup_{i\in\mathbb{N}} d_i) = f(\infty) = 1 \quad \neq \quad 0 = \bigsqcup_{i\in\mathbb{N}} f(d_i)$$

The monotone function f does not preserve the limits and thus it is not continuous.

5.2.2 Fixpoints

Now we are ready to study fixpoints of continuous functions.

Definition 5.15 (Pre-fixpoint and fixpoint). Let f be a continuous function on a $CPO_\perp$ $(D,\sqsubseteq)$. An element p is a *pre-fixpoint* if

$$f(p) \sqsubseteq p.$$

An element $d \in D$ is a *fixpoint* of f if

$$f(d) = d.$$

Of course any fixpoint of f is also a pre-fixpoint of f, i.e., the set of fixpoints of f is included in the set of its pre-fixpoints.

We will denote by $\mathrm{gfp}(f)$ the greatest fixpoint of f and by $\mathrm{lfp}(f)$ the least fixpoint of f, when they exist.

Let $f : D \to D$ and $d \in D$. We denote by $f^n(d)$ the repeated application of f to d for n times, i.e.,

$$f^0(d) \stackrel{\text{def}}{=} d$$
$$f^{n+1}(d) \stackrel{\text{def}}{=} f(f^n(d))$$

Lemma 5.2. *Let $(D, \sqsubseteq)$ be a partial order and $f : D \to D$ be a monotone function. The elements $\{f^n(\bot)\}_{n\in\mathbb{N}}$ form a chain in D.*

Proof. The property $\forall n \in \mathbb{N}.\ f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ can be readily proved by mathematical induction on n.

Base case: For $n = 0$ we have $f^0(\bot) = \bot \sqsubseteq f^1(\bot) = f(\bot)$, as $\bot$ is the least element of D.

Inductive case: Let us assume that the property holds for n, i.e., that

$$f^n(\bot) \sqsubseteq f^{n+1}(\bot).$$

We want to prove that the property holds for $n+1$, i.e., that

$$f^{n+1}(\bot) \sqsubseteq f^{n+2}(\bot).$$

In fact by definition we have $f^{n+1}(\bot) = f(f^n(\bot))$ and $f^{n+2}(\bot) = f(f^{n+1}(\bot))$. Since f is monotone and by the inductive hypothesis we have

$$f^{n+1}(\bot) = f(f^n(\bot)) \sqsubseteq f(f^{n+1}(\bot)) = f^{n+2}(\bot).$$

□

When $(D, \sqsubseteq)$ is a CPO then the chain $\{f^n(\bot)\}_{n\in\mathbb{N}}$ must have a limit $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$. The next theorem ensures that the least fixpoint of a continuous function always exists and that it is computed by the above limit.

Theorem 5.6 (Kleene's fixpoint theorem). *Let $f : D \to D$ be a continuous function on a CPO$_\bot$ D. Then, let*

$$\mathrm{fix}(f) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$$

The element $\mathrm{fix}(f) \in D$ has the following properties:

1. *$\mathrm{fix}(f)$ is a fixpoint of f, namely*

$$f(\mathrm{fix}(f)) = \mathrm{fix}(f)$$

2. $\mathrm{fix}(f)$ *is the least pre-fixpoint of* f*, namely*

$$f(d) \sqsubseteq d \Rightarrow \mathrm{fix}(f) \sqsubseteq d$$

Since any fixpoint is a pre-fixpoint, $\mathrm{fix}(f)$ *is also the least fixpoint of* f*.*

Proof. We prove the two items separately.

1. By continuity we will show that $\mathrm{fix}(f)$ is a fixpoint of f:

$$\begin{aligned} f(\mathrm{fix}(f)) &= f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) && \text{(by definition of fix)} \\ &= \bigsqcup_{n\in\mathbb{N}} f(f^n(\bot)) && \text{(by continuity of } f\text{)} \\ &= \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) && \text{(by definition of } f^{n+1}\text{)} \end{aligned}$$

   So we need to compute the limit of the chain

$$f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$$

   Since the limit is independent of any finite prefix of the chain, it coincides with the limit of the chain

$$f^0(\bot) = \bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$$

$$\begin{aligned} \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) &= \bigsqcup_{n\in\mathbb{N}} f^n(\bot) && \text{(by Lemma 5.1)} \\ &= \mathrm{fix}(f) && \text{(by definition of fix)} \end{aligned}$$

2. We want to prove that $\mathrm{fix}(f)$ is the least pre-fixpoint. We prove that any pre-fixpoint of f is an upper bound of the chain $\{f^n(\bot)\}_{n\in\mathbb{N}}$. Let d be a pre-fixpoint of f, i.e.,

$$f(d) \sqsubseteq d \tag{5.1}$$

   By mathematical induction we show that

$$\forall n \in \mathbb{N}. f^n(\bot) \sqsubseteq d$$

   i.e., that d is an upper bound for the chain $\{f^n(\bot)\}_{n\in\mathbb{N}}$:

   base case: obviously $f^0(\bot) = \bot \sqsubseteq d$
   inductive case: let us assume $f^n(\bot) \sqsubseteq d$. We want to prove that $f^{n+1}(\bot) \sqsubseteq d$:

$$\begin{aligned} f^{n+1}(\bot) &= f(f^n(\bot)) && \text{(by definition of } f^{n+1}\text{)} \\ &\sqsubseteq f(d) && \text{(by monotonicity of } f \\ & && \text{and inductive hypothesis)} \\ &\sqsubseteq d && \text{(because } d \text{ is a pre-fixpoint)} \end{aligned}$$

Since d is an upper bound for $\{f^n(\bot)\}_{n\in\mathbb{N}}$ and $\mathrm{fix}(f)$ is the limit (i.e., the least upper bound) of the same chain, it must be that $\mathrm{fix}(f) \sqsubseteq d$.

□

Now let us give two examples which show that the bottom element and the continuity property are required to compute the least fixpoint.

Example 5.18 (Bottom is necessary). Let $(\{\mathbf{true}, \mathbf{false}\}, \sqsubseteq)$ be the discrete order of boolean values. Obviously it is complete (because only finite chains of the form $x \sqsubseteq x \sqsubseteq x \sqsubseteq \ldots$ exist) and it has no bottom element, as **true** and **false** are not comparable. All functions over this domain are continuous. The identity function has two fixpoints, but there is no least fixpoint. On the contrary, the negation function has no fixpoint.

Example 5.19 (Continuity is necessary). Let us consider the $\mathrm{CPO}_\bot$ $(\mathbb{N} \cup \{\infty_1, \infty_2\}, \sqsubseteq)$ where

$$(\sqsubseteq \restriction \mathbb{N}) = \leq, \quad \forall d \in \mathbb{N} \cup \{\infty_1\}.\ d \sqsubseteq \infty_1, \quad \forall d \in \mathbb{N} \cup \{\infty_1, \infty_2\}.\ d \sqsubseteq \infty_2$$

The bottom element is 0. We define a monotone function f as follows (see Figure 5.5):

$$f(n) \stackrel{\text{def}}{=} \begin{cases} n+1 & \text{if } n \in \mathbb{N} \\ \infty_2 & \text{otherwise} \end{cases}$$

Note that f is not continuous. Let us consider the chain of even numbers $\{d_i\}_{i\in\mathbb{N}}$. It follows that $\{f(d_i)\}_{i\in\mathbb{N}}$ is the chain of odd numbers. We have

$$\bigsqcup_{i\in\mathbb{N}} d_i = \infty_1 \qquad \bigsqcup_{i\in\mathbb{N}} f(d_i) = \infty_1$$

Therefore:

$$f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = f(\infty_1) = \infty_2 \neq \infty_1 = \bigsqcup_{i\in\mathbb{N}} f(d_i)$$

Note that f has only one fixpoint, indeed

$$f(\infty_2) = \infty_2$$

But this fixpoint is not reachable by taking $\bigsqcup_{n\in\mathbb{N}} f^n(0) = \infty_1$.

5.3 Immediate Consequence Operator

In this section we reconcile two different approaches for defining semantics: inference rules, such as those used for defining the operational semantics of IMP, and fixpoint theory, which will be applied to define the denotational semantics of IMP. We show

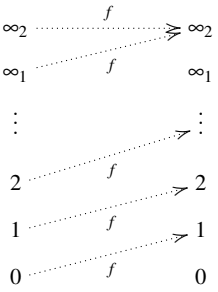


Fig. 5.5: Continuity is necessary

that the set of theorems of a logical system R can be defined as the least fixpoint of a suitable operator, called the *immediate consequence operator* and denoted $\widehat{R}$.

5.3.1 The Operator $\widehat{R}$

Let us consider a set F of well-formed formulas and a set R of inference rules over them. We define an operator $\widehat{R}$ over $(\wp(F), \subseteq)$, the $CPO_\perp$ of sets of well-formed formulas ordered by inclusion.

Definition 5.16 (Immediate consequence operator $\widehat{R}$). Let R be a logical system. We define a function $\widehat{R} : \wp(F) \to \wp(F)$ as follows (for any $S \subseteq F$):

$$\widehat{R}(S) \stackrel{\text{def}}{=} \{y \mid \exists (X/y) \in R.\ X \subseteq S\}$$

The function $\widehat{R}$ is called the *immediate consequence operator*.

The operator $\widehat{R}$, when applied to a set of well-formed formulas S, calculates a new set of formulas by applying the inference rules of R to the facts in S in all possible ways, i.e., $\widehat{R}(S)$ is the set of conclusions we can derive in one step from the hypotheses in S using rules in R. We will show that the set of theorems of R is equal to the least fixpoint of the immediate consequence operator $\widehat{R}$.

To apply the fixpoint theorem, we need to prove that $\widehat{R}$ is monotone and continuous.

Theorem 5.7 (Monotonicity of $\widehat{R}$). *$\widehat{R}$ is a monotone function.*

Proof. Let $S_1 \subseteq S_2$. We want to show that $\widehat{R}(S_1) \subseteq \widehat{R}(S_2)$. Let us assume $y \in \widehat{R}(S_1)$, then there exists a rule $(X/y) \in R$ with $X \subseteq S_1$. So we have $X \subseteq S_2$ and $y \in \widehat{R}(S_2)$. □

Theorem 5.8 (Continuity of $\widehat{R}$). *Let R be a logical system such that for any $(X/y) \in R$ the set of premises X is finite. Then $\widehat{R}$ is continuous.*

Proof. Let $\{S_i\}_{i\in\mathbb{N}}$ be a chain in $\wp(F)$. We want to prove that

$$\bigcup_{i\in\mathbb{N}} \widehat{R}(S_i) = \widehat{R}(\bigcup_{i\in\mathbb{N}} S_i).$$

As usual we prove the two inclusions separately:

$\subseteq$) Let $y \in \bigcup_{i\in\mathbb{N}} \widehat{R}(S_i)$ so there exists a natural number k such that $y \in \widehat{R}(S_k)$. Since

$$S_k \subseteq \bigcup_{i\in\mathbb{N}} S_i$$

by monotonicity

$$\widehat{R}(S_k) \subseteq \widehat{R}\left(\bigcup_{i\in\mathbb{N}} S_i\right)$$

hence $y \in \widehat{R}(\bigcup_{i\in\mathbb{N}} S_i)$.

$\supseteq$) Let $y \in \widehat{R}(\bigcup_{i\in\mathbb{N}} S_i)$ so there exists a rule $X/y \in R$ with $X \subseteq \bigcup_{i\in\mathbb{N}} S_i$. Since X is finite, there exists a natural number k such that $X \subseteq S_k$. In fact, for every $x \in X$ there will be a natural number k_x with $x \in S_{k_x}$ and letting $k = \max\{k_x\}_{x\in X}$ we have $X \subseteq S_k$. Since $X \subseteq S_k$ we have $y \in \widehat{R}(S_k) \subseteq \bigcup_{i\in\mathbb{N}} \widehat{R}(S_i)$ as required. □

5.3.2 Fixpoint of $\widehat{R}$

Now we are ready to present the fixpoint of $\widehat{R}$. For this purpose let us define I_R as the set of theorems provable in R:

$$I_R \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} I_R^i$$

where

$$I_R^0 \stackrel{\text{def}}{=} \varnothing$$
$$I_R^{n+1} \stackrel{\text{def}}{=} \widehat{R}(I_R^n) \cup I_R^n$$

Note that the generic I_R^n contains all theorems provable with derivations of depth[1] at most n, and I_R contains all theorems provable by using the rule system R.

Theorem 5.9. *Let R be a rule system. Then*

$$\forall n \in \mathbb{N}.\ I_R^n = \widehat{R}^n(\varnothing)$$

Proof. By induction on n.

[1] See Problem 4.12.

base case: $I_R^0 = \widehat{R}^0(\varnothing) = \varnothing$.

inductive case: We assume $I_R^n = \widehat{R}^n(\varnothing)$ and want to prove $I_R^{n+1} = \widehat{R}^{n+1}(\varnothing)$. Then

$$\begin{aligned} I_R^{n+1} &= \widehat{R}(I_R^n) \cup I_R^n && \text{(by definition of } I_R^{n+1}\text{)} \\ &= \widehat{R}(\widehat{R}^n(\varnothing)) \cup \widehat{R}^n(\varnothing) && \text{(by inductive hypothesis)} \\ &= \widehat{R}^{n+1}(\varnothing) \cup \widehat{R}^n(\varnothing) && \text{(by definition of } \widehat{R}^{n+1}\text{)} \\ &= \widehat{R}^{n+1}(\varnothing) && \text{(because } \widehat{R}^{n+1}(\varnothing) \supseteq \widehat{R}^n(\varnothing)\text{)} \end{aligned}$$

In the last step of the proof we have exploited the property $\widehat{R}^{n+1}(\varnothing) \supseteq \widehat{R}^n(\varnothing)$, which is an instance of Lemma 5.2 (by taking $D = \wp(F)$, $\sqsubseteq = \subseteq$, $\bot = \varnothing$ and $f = \widehat{R}$). □

Theorem 5.10 (Fixpoint of $\widehat{R}$). *Let R be a logical system. Then*

$$\mathrm{fix}(\widehat{R}) = I_R$$

Proof. By continuity of $\widehat{R}$ (Theorem 5.8) and the fixpoint theorem (Theorem 5.6), we know that the least fixpoint of $\widehat{R}$ exists and that

$$\mathrm{fix}(\widehat{R}) \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \widehat{R}^n(\varnothing)$$

Then, by Theorem 5.9:

$$I_R \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} I_R^n = \bigcup_{n \in \mathbb{N}} \widehat{R}^n(\varnothing) \stackrel{\text{def}}{=} \mathrm{fix}(\widehat{R})$$

as required. □

Example 5.20 (Rule system with discontinuous $\widehat{R}$). Let us consider the logical system R below:

$$\frac{\varnothing}{P(1)} \qquad \frac{P(x)}{P(x+1)} \qquad \frac{\forall n \in \mathbb{N}.\ P(1+2\times n)}{P(0)}$$

To ensure the continuity of $\widehat{R}$, Theorem 5.8 requires that the system has only rules with finitely many premises. The third rule of our system instead has infinitely many premises; it corresponds to

$$\frac{P(1) \quad P(3) \quad P(5) \quad \cdots}{P(0)}$$

The continuity of $\widehat{R}$, namely the fact that for all chains $\{S_i\}_{i \in \mathbb{N}}$ we have $\bigcup_{i \in \mathbb{N}} \widehat{R}(S_i) = \widehat{R}(\bigcup_{i \in \mathbb{N}} S_i)$, does not hold in this case. Indeed if we take the chain

$$\{P(1)\} \subseteq \{P(1), P(3)\} \subseteq \{P(1), P(3), P(5)\} \ldots$$

we have

i	0		1		2	$\cdots$
S_i	$\{P(1)\}$	$\subseteq$	$\{P(1),P(3)\}$	$\subseteq$	$\{P(1),P(3),P(5)\}$	$\cdots$
$\widehat{R}(S_i)$	$\{P(1),P(2)\}$	$\subseteq$	$\{P(1),P(2),P(4)\}$	$\subseteq$	$\{P(1),P(2),P(4),P(6)\}$	$\cdots$

from which we get

$$\begin{aligned}
\bigcup_{i\in\mathbb{N}} S_i &= \{P(1),P(3),P(5),\ldots\}\\
\widehat{R}(\bigcup_{i\in\mathbb{N}} S_i) &= \{P(1),P(2),P(4),\ldots \underbrace{P(0)}_{\text{3rd rule}}\}\\
\bigcup_{i\in\mathbb{N}} \widehat{R}(S_i) &= \{P(1),P(2),P(4),P(6),\ldots\}
\end{aligned}$$

because the third rule applies only when the predicate P holds for all the odd numbers, as in $\bigcup_{i\in\mathbb{N}} S_i$. Let us now compute the limit of $\widehat{R}$:

$$\text{fix}(\widehat{R}) = \bigcup_{n\in\mathbb{N}} \widehat{R}^n(\varnothing) = \{P(1),P(2),P(3),P(4),\ldots\}$$

In fact, we have

$$\begin{aligned}
\widehat{R}^0(\varnothing) &= \varnothing\\
\widehat{R}^1(\varnothing) &= \{P(1)\}\\
\widehat{R}^2(\varnothing) &= \{P(1),P(2)\}\\
\widehat{R}^3(\varnothing) &= \{P(1),P(2),P(3)\}\\
&\cdots
\end{aligned}$$

But $\text{fix}(\widehat{R})$ is not a fixpoint of $\widehat{R}$, because $P(0) \notin \text{fix}(\widehat{R})$ but $P(0) \in \widehat{R}(\text{fix}(\widehat{R}))$!

$$\widehat{R}(\text{fix}(\widehat{R})) = \{P(0),P(1),P(2),P(3),P(4),\ldots\} \neq \text{fix}(\widehat{R})$$

Example 5.21 (Balanced parentheses). Let us consider the grammar for balanced parentheses, from Example 2.5:

$$S \quad ::= \quad \varepsilon \quad | \quad (S) \quad | \quad SS$$

The corresponding logical system is:

$$\frac{\varnothing}{\varepsilon \in L_S} \qquad \frac{s \in L_S}{(s) \in L_S} \qquad \frac{s_1 \in L_S \quad s_2 \in L_S}{s_1 s_2 \in L_S}$$

So we can use the $\widehat{R}$ operator and the fixpoint theorem to find all the strings generated by the grammar by letting $L_S = \text{fix}(\widehat{R})$:

$$
\begin{aligned}
L_{S0} &= \widehat{R}^0(\varnothing) = \varnothing \\
L_{S1} &= \widehat{R}(S_0) = \{\varepsilon\} \\
L_{S2} &= \widehat{R}(S_1) = \{\varepsilon,\ (\,)\} \\
L_{S3} &= \widehat{R}(S_2) = \{\varepsilon,\ (\,),\ ((\,)),\ (\,)(\,)\} \\
&\ldots
\end{aligned}
$$

Problems

5.1. Prove Theorem 5.1. *Hint:* The proof is easy, because the axioms of partial and total orders are all universally quantified.

5.2. Let $(\wp(\mathbb{N}), \subseteq)$ be the $\mathrm{CPO}_\perp$ of sets of natural numbers, ordered by inclusion. Assume a set $X \subseteq \mathbb{N}$ is fixed. Let $f, g : \wp(\mathbb{N}) \longrightarrow \wp(\mathbb{N})$ be the functions

$$
\begin{aligned}
f(S) &\stackrel{\text{def}}{=} S \cap X \\
g(S) &\stackrel{\text{def}}{=} (\mathbb{N} \setminus S) \cap X
\end{aligned}
$$

1. Are f and g monotone? Are they continuous?
2. Do the answers to the above questions depend on the given set X?

5.3. Define three functions $f_i : D_i \to D_i$ over three suitable CPOs D_i for $i \in [1,3]$ (not necessarily with bottom) such that

1. f_1 is continuous, has fixpoints but not a least fixpoint;
2. f_2 is continuous and has no fixpoint;
3. f_3 is monotone but not continuous.

5.4. Define a partial order $\mathscr{D} = (D, \sqsubseteq)$ that is not complete.

1. Let $x \prec y$ be the irreflexive relation obtained by reversing the order, i.e.

$$x \prec y \quad \text{if and only if} \quad y \sqsubseteq x \wedge x \neq y.$$

   Is $\mathscr{D}' = (D, \prec)$ a well-founded relation?
2. In general, is it possible that $\mathscr{D}'$ is well-founded for some $\mathscr{D}$?

5.5. Let $V^* \cup V^\infty$ be the set of finite (V^*) and infinite (V^∞) strings over the alphabet $V = \{a, b, c\}$, and let $\alpha \sqsubseteq \alpha\beta$, where juxtaposition in $\alpha\beta$ denotes string concatenation and $\alpha\beta = \alpha$ if α is infinite.

1. Is the structure $(V^* \cup V^\infty, \sqsubseteq)$ a partial order?
2. If yes, is it a complete partial order?
3. Does there exist a bottom element?
4. Which are the maximal elements?

5.6. Let $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ be two CPOs such that $D_1, D_2 \subseteq D$. Consider the structures

- $(D_1 \cup D_2\ ,\ \sqsubseteq)$, where $x \sqsubseteq y$ iff $x \sqsubseteq_1 y \vee x \sqsubseteq_2 y$
- $(D_1 \cap D_2\ ,\ \preceq)$, where $x \preceq y$ iff $x \sqsubseteq_1 y \wedge x \sqsubseteq_2 y$

1. Are they always partial orders?
2. If so, are they complete?

In the case of negative answers, exhibit some counterexample.

5.7. Let X and Y be sets and $X_\perp$ and $Y_\perp$ be the corresponding flat domains. Show that a function $f : X_\perp \to Y_\perp$ is continuous if and only if one, or both, of the following conditions holds:

1. f is *strict*, i.e., $f(\perp) = \perp$.
2. f is constant.

5.8. Let $\{\top\}$ be a one-element set and $\{\top\}_\perp$ the corresponding flat domain. Let Ω be the domain of *vertical natural numbers* (see Examples 5.6 and 5.11)

$$0 \leq 1 \leq 2 \leq 3 \leq \ldots \leq \infty.$$

Show that the set of continuous functions from Ω to $\{\top\}_\perp$ is in bijection with Ω. *Hint:* Define the possible continuous functions from Ω to $\{\top\}_\perp$.

5.9. Let $D = \{n \in \mathbb{N} \mid n > 0\} \cup \{\infty_0\}$ and $\sqsubseteq$ be the relation over D such that

- for any pair of natural numbers $n, m \in D$, we let $n \sqsubseteq m$ iff n divides m;
- for any $x \in D$, we let $x \sqsubseteq \infty_0$.

Is $(D, \sqsubseteq)$ a $\mathrm{CPO}_\perp$? Explain.

5.10. Consider the set $\mathbb{N} \times \mathbb{N}$ of pairs of natural numbers with the lexicographic order relation $\sqsubseteq$ defined by letting

$$(n, m) \sqsubseteq (n', m') \quad \text{if} \quad n < n' \vee (n = n' \wedge m < m')$$

1. Prove that $\sqsubseteq$ is a partial order with bottom.
2. Show that the chain $\{(0, k)\}_{k \in \mathbb{N}}$ has a lub.
3. Exhibit a chain without lub.
4. Consider the subset $[0, n] \times \mathbb{N}$, with the same order, and then show, also in this case, a chain without lub .
5. Finally, prove that $[0, n] \times (\mathbb{N} \cup \{\infty\})$ with the same order (where $x \leq \infty$ for any $x \in \mathbb{N}$), is complete with bottom, and show a monotone, non-continuous function on it.

5.11. Prove that the set **Tf** of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ defined in Example 5.14 forms a complete partial order.

5.12. Consider the set **PI** of partial injective functions from $\mathbb{N}$ to $\mathbb{N}$. A partial injective function f can be seen as a relation $\{(x,y) \mid x,y \in \mathbb{N} \wedge y = f(x)\} \subseteq \mathbb{N} \times \mathbb{N}$ such that

- $(x,y),(x,y') \in f$ implies $y = y'$, (i.e., f is a partial function), and
- $(x,y),(x',y) \in f$ implies $x = x'$, (i.e., f is injective).

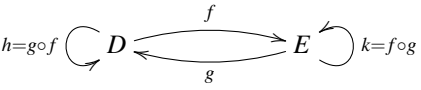
Accordingly, the elements of **PI** can be ordered by inclusion.

1. Prove that $(\mathbf{PI}, \subseteq)$ is a complete partial order.
2. Prove that the function $F : \mathbf{PI} \to \mathbf{PI}$ with $F(f) = \{(2 \times x, y) \mid (x,y) \in f\}$ is monotone and continuous.
   (*Hint*: Consider F as computed by the immediate consequences operator $\widehat{R}$, with R consisting only of the rule $(x,y)/(2 \times x, y)$.)

5.13. Let $(D, \sqsubseteq)$ be a CPO, $\{d_i\}_{i \in \mathbb{N}}$ a chain in D and $f : \mathbb{N} \to \mathbb{N}$ a function such that for all $i, j \in \mathbb{N}$ if $i < j$ then $f(i) < f(j)$. Prove that

$$\bigsqcup_{i \in \mathbb{N}} d_{f(i)} = \bigsqcup_{i \in \mathbb{N}} d_i$$

5.14. Let D, E be two $\mathrm{CPO}_\perp$s and $f : D \to E$, $g : E \to D$ be two continuous functions between them. Their compositions $h = g \circ f : D \to D$ and $k = f \circ g : E \to E$ are known to be continuous and thus have least fixpoints.



Let $e_0 = \mathrm{fix}(k) \in E$. Prove that $g(e_0) = \mathrm{fix}(h) \in D$ by showing that

1. $g(e_0)$ is a fixpoint for h, and
2. $g(e_0)$ is the least pre-fixpoint for h.

Chapter 6
Denotational Semantics of IMP

The point is that, mathematically speaking, functions are independent of their means of computation and hence are "simpler" than the explicitly generated, step-by-step evolved sequences of operations on representations. (Dana Scott)

Abstract In this chapter we give a more abstract, purely mathematical semantics to IMP, called *denotational semantics*. The operational semantics is close to the memory-based, executable machine-like view: given a program and a state, we derive the state obtained after the execution of that program. The denotational semantics takes a program and returns the transformation function over memories associated with that program: given an initial state as the argument, the final state is returned as the result. Since functions will be written in some fixed mathematical notation, i.e., they can also be regarded as "programs" of a suitable formalism, we can say that, to some extent, the operational semantics defines an "interpreter" of the language (given a program *and* the initial state it returns the final state obtained by executing the program), while the denotational semantics defines a "compiler" for the language (from programs to functions, i.e., programs written in a more abstract language). We conclude the chapter by reconciling the equivalences induced by the operational and the denotational semantics and by stating the principle of computational induction.

6.1 λ-Notation

In the following we shall rely on λ*-notation* as a *(meta-)language* for writing anonymous functions. When considering HOFL, λ-notation will be used both at the level of the programming language and at the level of the denotational semantics, as a meta-language.

The λ-calculus was introduced by Alonzo Church (1903–1995) in order to answer one of the questions posed by David Hilbert (1862–1943) in his program, known as the Entscheidungsproblem (German for *decision problem*). Roughly, the problem consisted in the existence of an algorithm to decide whether a given statement of a first-order logic (possibly enriched with a finite number of axioms) is deducible or not from the axioms of logic. Alan Turing (1912–1954) proved that no effectively calculable algorithm can exist that solves the problem, where "calculable" meant

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_6

computable by a Turing machine. Independently, Alonzo Church answered negatively assuming that "calculable" meant a function expressible in the λ-calculus.

6.1.1 λ-Notation: Main Ideas

The λ-calculus is built around the idea of expressing a calculus of functions, where it is not necessary to assign names to functions, i.e., where functions can be expressed anonymously. Conceptually, this amounts to the possibility of

- forming (anonymous) functions by *abstraction* over names in an expression; and
- *applying* a function to an argument.

Building on the two basic considerations above, Church developed a theory of functions based on rules for computation, as opposed to the classical set-theoretic view of functions as sets of pairs (argument, result).

Example 6.1. Let us start with a simple example from arithmetic. Take a polynomial such as

$$x^2 - 2x + 5$$

What is the value of the above expression when x is replaced by 2? We compute the result by plugging in '2' for 'x' in the expression to get

$$2^2 - 2 \times 2 + 5 = 5$$

In λ-notation, when we want to express that the value of an expression depends on some value to be plugged in, we use abstraction. Syntactically, this corresponds to prefixing the expression by the special symbol λ and the name of the formal parameter, as, e.g., in

$$\lambda x.\ (x^2 - 2x + 5)$$

The informal reading is

> wait for a value v to replace x and then compute $v^2 - 2v + 5$.

We want to be able to pass some actual parameter to the function above, i.e., to apply the function to some value v. To this aim, we denote application by juxtaposition:

$$(\lambda x.\ (x^2 - 2x + 5))\ 2$$

means that the function $(\lambda x.\ (x^2 - 2x + 5))$ is applied to 2 (i.e., that the actual parameter 2 must replace the occurrences of the formal parameter x in $x^2 - 2x + 5$, to obtain $2^2 - 2 \times 2 + 5 = 5$).

Note that

- by writing $\lambda x.\ t$ we are declaring x as a formal parameter appearing in t;
- the symbol λ has no particular meaning (any other symbol could have been used);

- we say that λx 'binds' the (occurrences of the) variable x in t;
- the scope of the formal parameter x is just t; if x occurs also "outside" t, then it refers to another (homonymous) identifier.

Example 6.2. Let us consider another example:

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2$$

This time we have a function that is waiting for two arguments (first x, then y), but to which we pass one value (2). We have

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2 = \lambda y.\ (2^2 - 2y + 5) = \lambda y.\ (9 - 2y)$$

That is, the result of applying $\lambda x.\ \lambda y.\ (x^2 - 2y + 5)$ to 2 is the function $(\lambda y.\ (9 - 2y))$.

In the λ-calculus we can pass functions as arguments and return functions as results.

Example 6.3. Take the term $\lambda f.\ (f\ 2)$: it waits for a function f that will be applied to the value 2. If we pass the function $(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$ to $\lambda f.\ (f\ 2)$, written

$$(\lambda f.\ (f\ 2))\ (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$$

then we get the function $\lambda y.\ (9 - 2y)$ as a result.

Definition 6.1 (Lambda terms). We define *lambda terms* as the terms generated by the grammar

$$t \quad ::= \quad x \quad | \quad \lambda x.\, t \quad | \quad (t_0\ t_1) \quad | \quad t \to (t_0, t_1)$$

where x is a variable.

As we can see the lambda notation is very simple. It has four constructs:

- x: is a simple *variable*.
- $\lambda x.\ t$: is the *lambda abstraction* which allows us to define anonymous functions.
- $t_0\ t_1$: is the *application* of a function t_0 to its argument t_1.
- $t \to t_0, t_1$ is the conditional operator, i.e. the "if-then-else" construct in lambda notation.

Note that we omit some parentheses when no ambiguity can arise.

Lambda abstraction $\lambda x.\ t$ is the main feature. It allows us to define functions, where x represents the parameter of the function and t is the lambda term which represents the body of the function. For example the term $\lambda x.\ x$ is the identity function.

Note that while we can have different terms t and t' that define the same function, Church proved that the problem of deciding whether $t = t'$ is undecidable.

Definition 6.2 (Conditional expressions). Let t, t_0 and t_1 be three lambda terms. We define

$$t \to t_0, t_1 = \begin{cases} t_0 & \text{if } t = true \\ t_1 & \text{if } t = false \end{cases}$$

All the notions used in this definition, such as "true" and "false", can be formalised in lambda notation only, by using lambda abstraction, as shown in Section 6.1.1.1 for the interested reader. In the following we will take the liberty of assuming that data types such as integers and booleans are available in the lambda-notation as well as the usual operations on them.

Remark 6.1 (Associativity of abstraction and application). In the following, to limit the number of parentheses and keep the notation more readable, we assume that application is left-associative, and lambda abstraction is right-associative, i.e.,

$$\begin{array}{ll} t_1\ t_2\ t_3\ t_4 & \text{is read as } (((t_1\ t_2)\ t_3)\ t_4) \\ \lambda x_1.\ \lambda x_2.\ \lambda x_3.\ \lambda x_4.\ t & \text{is read as } \lambda x_1.\ (\lambda x_2.\ (\lambda x_3.\ (\lambda x_4.\ t))) \end{array}$$

Remark 6.2 (Precedence of application). We will also assume that application has precedence over abstraction, i.e.,

$$\lambda x.\ t\ t' = \lambda x.\ (t\ t')$$

6.1.1.1 λ-Notation: Booleans and Church Numerals

In the above examples, we have enriched standard arithmetic expressions with abstraction and application. In general, it would be possible to encode booleans and numbers (and operations over them) just using abstraction and application.

For example, let us consider the following terms:

$$T \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x$$
$$F \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ y$$

We can assume that T represents true and F represents false.

Under this convention, we can define the usual logical operations by letting

$$\text{AND} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ q\ p$$
$$\text{OR} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ p\ q$$
$$\text{NOT} \stackrel{\text{def}}{=} \lambda p.\ \lambda x.\ \lambda y.\ p\ y\ x$$

Now suppose that P will reduce either to T or to F. The expression $P\ A\ B$ can be read as 'if P then A else B'.

For natural numbers, we can adopt the convention that the number n is represented by a function that takes a function f and an argument x and applies f to x for n times consecutively. For example

$$0 \stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, x \qquad 1 \stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, f\, x \qquad 2 \stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, f\, (f\, x) \qquad \cdots$$

Then, the operations for successor, sum and multiplication can be defined by letting

$$\begin{aligned} \text{SUCC} &\stackrel{\text{def}}{=} \lambda n.\, \lambda f.\, \lambda x.\, f\, (n\, f\, x) \\ \text{SUM} &\stackrel{\text{def}}{=} \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, m\, f\, (n\, f\, x) \\ \text{MUL} &\stackrel{\text{def}}{=} \lambda n.\, \lambda m.\, \lambda f.\, n\, (m\, f) \end{aligned}$$

6.1.2 Alpha-Conversion, Beta-Rule and Capture-Avoiding Substitution

The names of the formal parameters we choose for a given function should not matter. Therefore, any two expressions that differ just in the particular choice of λ-abstracted variables and have the same structure otherwise should be considered be equal.

For example, we do not want to distinguish between the terms

$$\lambda x.\, (x^2 - 2x + 5) \qquad\qquad \lambda y.\, (y^2 - 2y + 5)$$

On the other hand, the expressions

$$x^2 - 2x + 5 \qquad\qquad y^2 - 2y + 5$$

must be distinguished, because depending on the context where they are used, the symbols x and y could have different meanings.

We say that two terms are α-convertible if one is obtained from the other by renaming some λ-abstracted variables. We call *free* the variables x whose occurrences are not inside the scope of a λ binding.

Definition 6.3 (Free variables). The set of free variables occurring in a term is defined by structural recursion:

$$\begin{aligned} \text{fv}(x) &\stackrel{\text{def}}{=} \{x\} \\ \text{fv}(\lambda x.\, t) &\stackrel{\text{def}}{=} \text{fv}(t) \setminus \{x\} \\ \text{fv}(t_0\, t_1) &\stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1) \\ \text{fv}(t \to t_0, t_1) &\stackrel{\text{def}}{=} \text{fv}(t) \cup \text{fv}(t_0) \cup \text{fv}(t_1) \end{aligned}$$

The second equation highlights that lambda abstraction is a binding operator.

Definition 6.4 (Alpha-conversion). We define *α-conversion* as the equivalence induced by letting

$$\lambda x.\, t = \lambda y.\, (t[{}^{y}/_{x}]) \quad \text{if } y \notin \mathrm{fv}(t)$$

where $t[{}^{y}/_{x}]$ denotes the substitution of x with y applied to the term t.

Note the side condition $y \notin \mathrm{fv}(t)$, which is needed to avoid 'capturing' other free variables appearing in t.

For example

$$\lambda z.\, z^2 - 2y + 5 = \lambda x.\, x^2 - 2y + 5 \neq \lambda y.\, y^2 - 2y + 5$$

We now have all the ingredients to define the basic computational rule, called the β-rule, which explains how to apply a function to an argument.

Definition 6.5 (Beta-rule). Let t, t' be two lambda terms. We define

$$(\lambda x.\, t')\, t = t'[{}^{t}/_{x}]$$

This axiom is called the β-rule.

In defining alpha-conversion and the beta-rule we have used substitutions such as $[{}^{y}/_{x}]$ and $[{}^{t}/_{x}]$. Let us now try to formalise the notion of substitution by structural recursion. What is wrong with the following naive attempt?

$$y[{}^{t}/_{x}] \stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(\lambda y.\, t')[{}^{t}/_{x}] \stackrel{\text{def}}{=} \begin{cases} \lambda y.\, t' & \text{if } y = x \\ \lambda y.\, (t'[{}^{t}/_{x}]) & \text{if } y \neq x \end{cases}$$

$$(t_0\, t_1)[{}^{t}/_{x}] \stackrel{\text{def}}{=} (t_0[{}^{t}/_{x}])\, (t_1[{}^{t}/_{x}])$$

$$(t' \to t_0, t_1)[{}^{t}/_{x}] \stackrel{\text{def}}{=} (t'[{}^{t}/_{x}]) \to (t_0[{}^{t}/_{x}]), (t_1[{}^{t}/_{x}])$$

Example 6.4 (Substitution, without alpha-renaming). Consider the terms

$$t \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, (x^2 - 2y + 5) \qquad t' \stackrel{\text{def}}{=} y$$

and apply t to t':

$$\begin{aligned} t\, t' &= (\lambda x.\, \lambda y.\, (x^2 - 2y + 5))\, y \\ &= (\lambda y.\, (x^2 - 2y + 5))[{}^{y}/_{x}] \\ &= \lambda y.\, ((x^2 - 2y + 5)[{}^{y}/_{x}]) \\ &= \lambda y.\, (y^2 - 2y + 5) \end{aligned}$$

It happens that the free variable $y \in \mathrm{fv}(t\ t')$ has been 'captured' by the lambda abstraction λy. Instead, free variables occurring in t should remain free during the application of the substitution $[^t/_x]$.

Thus we need to correct the above version of substitution for the case related to $(\lambda y.\ t')[^t/_x]$ by first applying the alpha-conversion to $\lambda y.\ t'$ (to make sure that if $y \in \mathrm{fv}(t)$, then the free occurrences of y in t will not be captured by λy when replacing x in t') and then the substitution $[^t/_x]$.

Definition 6.6 (Capture-avoiding substitution). Let t, t', t_0 and t_1 be four lambda terms. We define

$$
\begin{aligned}
y[^t/_x] &\stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(\lambda y.\ t')[^t/_x] &\stackrel{\text{def}}{=} \lambda z.\ ((t'[^z/_y])[^t/_x]) \quad \text{if } z \notin \mathrm{fv}(\lambda y.\ t') \cup \mathrm{fv}(t) \cup \{x\} \\
(t_0\ t_1)[^t/_x] &\stackrel{\text{def}}{=} (t_0[^t/_x])\ (t_1[^t/_x]) \\
(t' \to t_0, t_1)[^t/_x] &\stackrel{\text{def}}{=} (t'[^t/_x]) \to (t_0[^t/_x]), (t_1[^t/_x])
\end{aligned}
$$

Note that the matter of names is not so trivial. In the second equation we first rename y in t' with a fresh name z, then proceed with the substitution of x with t. As explained, this solution is motivated by the fact that y might not be free in t, but it introduces some nondeterminism in the equations due to the arbitrary nature of the new name z. This nondeterminism immediately disappears if we regard the terms up to the alpha-conversion equivalence, as previously introduced. Obviously α-conversion and substitution should be defined at the same time to avoid circularity. By using the α-conversion we can prove statements such as $\lambda x.\ x = \lambda y.\ y$.

Example 6.5 (Application with alpha-renaming). Consider the terms t, t' from Example 6.4. By exploiting Definition 6.6, we have

$$
\begin{aligned}
t\ t' &= (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ y \\
&= (\lambda y.\ (x^2 - 2y + 5))[^y/_x] \\
&= \lambda z.\ ((x^2 - 2y + 5)[^z/_y][^y/_x]) \\
&= \lambda z.\ ((x^2 - 2z + 5)[^y/_x]) \\
&= \lambda z.\ (y^2 - 2z + 5)
\end{aligned}
$$

Finally we introduce some notational conventions for omitting parentheses when defining the domains and codomains of functions:

$$
\begin{aligned}
A \to B \times C &= A \to (B \times C) & \qquad A \times B \times C &= (A \times B) \times C \\
A \times B \to C &= (A \times B) \to C & \qquad A \to B \to C &= A \to (B \to C)
\end{aligned}
$$

6.2 Denotational Semantics of IMP

As we said, we will use lambda notation as a meta-language; this means that we will express the semantics of IMP by translating IMP syntax into lambda terms.

The denotational semantics of IMP consists of three separate *interpretation* functions, one for each syntax category ($Aexp, Bexp, Com$):

Aexp: each arithmetic expression is mapped to a function from states to integers:

$$\mathscr{A} : Aexp \to (\Sigma \to \mathbb{Z})$$

Bexp: each boolean expression is mapped to a function from states to booleans:

$$\mathscr{B} : Bexp \to (\Sigma \to \mathbb{B})$$

Com: each command is mapped to a (partial) function from states to states:

$$\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$$

6.2.1 Denotational Semantics of Arithmetic Expressions: The Function $\mathscr{A}$

We shall define $\mathscr{A}$ by structural recursion over the syntax of arithmetic expressions. Let us fix some notation. We will rely on definitions of the form

$$\mathscr{A}\,[\![n]\!] \stackrel{\text{def}}{=} \lambda\sigma.\, n$$

with the following meaning:

- $\mathscr{A} : Aexp \to \Sigma \to \mathbb{Z}$ is the interpretation function,
- n is an arithmetic expression (i.e., a term in $Aexp$). The surrounding brackets $[\![$ and $]\!]$ emphasise that it is a piece of syntax rather then part of the meta-language.
- the expression $\mathscr{A}\,[\![n]\!]$ is a function whose type is $\Sigma \to \mathbb{Z}$. Notice that also the right part of the equation must be of the same type $\Sigma \to \mathbb{Z}$.

We shall often define the interpretation function $\mathscr{A}$ by writing equalities such as

$$\mathscr{A}\,[\![n]\!]\,\sigma \stackrel{\text{def}}{=} n$$

instead of

$$\mathscr{A}\,[\![n]\!] \stackrel{\text{def}}{=} \lambda\sigma.\, n$$

In this way, we simplify the notation in the right-hand side. Notice that both sides of the equation ($\mathscr{A}\,[\![n]\!]\,\sigma$ and n) have type $\mathbb{Z}$.

Definition 6.7 (Denotational semantics of arithmetic expressions). The denotational semantics of arithmetic expressions is defined by structural recursion as

$$\begin{aligned}
\mathscr{A}[\![n]\!]\,\sigma &\stackrel{\text{def}}{=} n \\
\mathscr{A}[\![x]\!]\,\sigma &\stackrel{\text{def}}{=} \sigma x \\
\mathscr{A}[\![a_0 + a_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\,\sigma) + (\mathscr{A}[\![a_1]\!]\,\sigma) \\
\mathscr{A}[\![a_0 - a_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\,\sigma) - (\mathscr{A}[\![a_1]\!]\,\sigma) \\
\mathscr{A}[\![a_0 \times a_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\,\sigma) \times (\mathscr{A}[\![a_1]\!]\,\sigma)
\end{aligned}$$

Let us briefly comment on the above definitions.

Constants:	The denotational semantics of any constant n is just the constant function that always returns n for any σ.
Variables:	The denotational semantics of any variable x is the function that takes a memory σ and returns the value of x in σ.
Binary expressions:	The denotational semantics of any binary expression evaluates the arguments (with the same given σ) and combines the results by exploiting the corresponding arithmetic operation.

Note that the symbols $+$, $-$ and $\times$ are overloaded: in the left-hand side they represent elements of the syntax, while in the right-hand side they represent operators of the meta-language. Similarly for the symbol n in the first equation.

6.2.2 Denotational Semantics of Boolean Expressions: The Function $\mathscr{B}$

The denotational semantics of boolean expressions is given by a function $\mathscr{B}$ defined in a very similar way to $\mathscr{A}$. The only differences are that the values to be returned are elements of $\mathbb{B}$ and not of $\mathbb{Z}$ and that $\mathscr{B}$ is not always defined in terms of itself: some defining equations exploit the function $\mathscr{A}$.

Definition 6.8 (Denotational semantics of boolean expressions). The denotational semantics of boolean expressions is defined by structural recursion as follows:

$$\begin{aligned}
\mathscr{B}[\![v]\!]\,\sigma &\stackrel{\text{def}}{=} v \\
\mathscr{B}[\![a_0 = a_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\,\sigma) = (\mathscr{A}[\![a_1]\!]\,\sigma) \\
\mathscr{B}[\![a_0 \leq a_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\,\sigma) \leq (\mathscr{A}[\![a_1]\!]\,\sigma) \\
\mathscr{B}[\![\neg b]\!]\,\sigma &\stackrel{\text{def}}{=} \neg\,(\mathscr{B}[\![b]\!]\,\sigma) \\
\mathscr{B}[\![b_0 \vee b_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{B}[\![b_0]\!]\,\sigma) \vee (\mathscr{B}[\![b_1]\!]\,\sigma) \\
\mathscr{B}[\![b_0 \wedge b_1]\!]\,\sigma &\stackrel{\text{def}}{=} (\mathscr{B}[\![b_0]\!]\,\sigma) \wedge (\mathscr{B}[\![b_1]\!]\,\sigma)
\end{aligned}$$

6.2.3 Denotational Semantics of Commands: The Function $\mathscr{C}$

We are now ready to present the denotational semantics of commands. As one might expect, the interpretation function of commands is the most complex. It has the following type:

$$\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$$

Since commands can diverge, the codomain of $\mathscr{C}$ is the set of partial functions from memories to memories. As we have discussed in Example 5.14, for each partial function we can define an equivalent total function. So we define

$$\mathscr{C} : Com \to (\Sigma \to \Sigma_\perp)$$

This will simplify the notation.

Instead of presenting the whole, structurally recursive, definition of $\mathscr{C}$ and then discussing its defining equations, we give each rule separately accompanied by the necessary explanations.

We start from the simplest commands: skip and assignments.

$$\mathscr{C}[\![\mathbf{skip}]\!]\,\sigma \stackrel{\text{def}}{=} \sigma \tag{6.1}$$

We see that $\mathscr{C}[\![\mathbf{skip}]\!]$ is the identity function: **skip** does not modify the memory.

$$\mathscr{C}[\![x := a]\!]\,\sigma \stackrel{\text{def}}{=} \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}] \tag{6.2}$$

The denotational semantics of the assignment evaluates the arithmetic expression a via $\mathscr{A}$ and then modifies the memory by assigning the corresponding value to the location x.

Let us now consider the sequential composition of two commands. In interpreting $c_0; c_1$ we first interpret c_0 in the starting memory and then c_1 in the state produced by c_0. The problem is that from the first application of $\mathscr{C}[\![c_0]\!]$ we obtain a value in $\Sigma_\perp$, not necessarily in Σ, so we cannot apply $\mathscr{C}[\![c_1]\!]$. To work this problem out we introduce a *lifting* operator $(\cdot)^*$: it takes a function in $\Sigma \to \Sigma_\perp$ and returns a function in $\Sigma_\perp \to \Sigma_\perp$, i.e., its type is $(\Sigma \to \Sigma_\perp) \to (\Sigma_\perp \to \Sigma_\perp)$.

Definition 6.9 (Lifting). Let $f : \Sigma \to \Sigma_\perp$. We define a function $f^* : \Sigma_\perp \to \Sigma_\perp$ as follows:

$$f^*(x) = \begin{cases} \perp & \text{if } x = \perp \\ f(x) & \text{otherwise} \end{cases}$$

So the definition of the interpretation function for $c_0; c_1$ is

$$\mathscr{C}[\![c_0; c_1]\!]\,\sigma \stackrel{\text{def}}{=} \mathscr{C}[\![c_1]\!]^*\,(\mathscr{C}[\![c_0]\!]\,\sigma) \tag{6.3}$$

Note that we apply the lifted version $\mathscr{C}[\![c_1]\!]^*$ of $\mathscr{C}[\![c_1]\!]$ to the argument $\mathscr{C}[\![c_0]\!]\,\sigma$.

Let us now consider the conditional command. Recall that the λ-calculus provides a conditional operator; then we have immediately

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \tag{6.4}$$

The definition of the denotational semantics of the while command is more intricate. We could think to define the interpretation simply as

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

Obviously this definition is not a structural recursion, because the same expression $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ whose meaning we want to define appears in the right-hand side of the defining equation. Indeed structural recursion allows only for the presence of subterms in the right-hand side, like $\mathscr{B}[\![b]\!]$ and $\mathscr{C}[\![c]\!]$. To solve this issue we will reduce the problem of defining the semantics of iteration to a fixpoint calculation. Let us define a function $\Gamma_{b,c} : (\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$:

$$\Gamma_{b,c} \stackrel{\text{def}}{=} \lambda\varphi.\ \underbrace{\lambda\sigma.\ \underbrace{\underbrace{\mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma}_{\Sigma_\perp}}_{\Sigma \to \Sigma_\perp}}_{(\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp}$$

The function $\Gamma_{b,c}$ takes a function $\varphi : \Sigma \to \Sigma_\perp$, and returns the function

$$\lambda\sigma.\ \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

of type $\Sigma \to \Sigma_\perp$, which given a memory σ evaluates $\mathscr{B}[\![b]\!]\sigma$ and depending on the outcome returns either $\varphi^*(\mathscr{C}[\![c]\!]\sigma)$ or σ. Note that the definition of $\Gamma_{b,c}$ refers only to subterms of the command **while** b **do** c. Clearly we require that $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ is a fixpoint of $\Gamma_{b,c}$, i.e., that

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] = \Gamma_{b,c}\,\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$$

As there can be several fixpoints for $\Gamma_{b,c}$, we define $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ to be the least one. Next we show that $\Gamma_{b,c}$ is a monotone and continuous function, so that we can prove that $\Gamma_{b,c}$ has a least fixpoint and that by the fixpoint Theorem 5.6

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma_{b,c} = \bigsqcup_{n \in \mathbb{N}} \Gamma_{b,c}^n(\perp_{\Sigma \to \Sigma_\perp}) \tag{6.5}$$

To prove continuity we will consider $\Gamma_{b,c}$ as operating on partial functions:

$$\Gamma_{b,c} : (\Sigma \rightharpoonup \Sigma) \longrightarrow (\Sigma \rightharpoonup \Sigma).$$

Partial functions in $\Sigma \rightharpoonup \Sigma$ can be represented as sets of pairs (σ, σ') that we write as formulas $\sigma \mapsto \sigma'$. Then the effect of $\Gamma_{b,c}$ can be represented by the immediate consequence operators for the following set of rules:

$$R_{\Gamma_{b,c}} \stackrel{\text{def}}{=} \left\{ \frac{\mathscr{B}[\![b]\!]\sigma \quad \mathscr{C}[\![c]\!]\sigma = \sigma'' \quad \sigma'' \mapsto \sigma'}{\sigma \mapsto \sigma'}, \qquad \frac{\neg\mathscr{B}[\![b]\!]\sigma}{\sigma \mapsto \sigma} \right\}$$

Note that there are infinitely many instances of the rules, but each rule has only a finite number of premises, and that

$$\widehat{R}_{\Gamma_{b,c}} = \Gamma_{b,c}.$$

The only formulas appearing in the rules are $\sigma'' \mapsto \sigma'$ (as a premise of the first rule), $\sigma \mapsto \sigma'$ and $\sigma \mapsto \sigma$ (as conclusions); the other formulas express side conditions: $\mathscr{B}[\![b]\!]\sigma \wedge \mathscr{C}[\![c]\!]\sigma = \sigma''$ for the first rule and $\neg\mathscr{B}[\![b]\!]\sigma$ for the second rule. An instance of the first rule schema is obtained by picking two memories σ and σ'' such that $\mathscr{B}[\![b]\!]\sigma$ is true and $\mathscr{C}[\![c]\!]\sigma = \sigma''$. Then for every σ' such that $\sigma'' \mapsto \sigma'$ we can derive $\sigma \mapsto \sigma'$. The second rule schema is an axiom expressing that $\sigma \mapsto \sigma$ whenever $\neg\mathscr{B}[\![b]\!]\sigma$.

Since all the rules obtained in this way have a finite number of premises (actually one or none), we can apply Theorem 5.8, which ensures the continuity of $\widehat{R}_{\Gamma_{b,c}}$. Now by using Theorem 5.10 we have

$$\text{fix } \Gamma_{b,c} = \text{fix } \widehat{R}_{\Gamma_{b,c}} = I_{R_{\Gamma_{b,c}}}$$

Let us conclude this section with three examples which explain how to use the definitions we have given.

Example 6.6. Let us consider the command

$$w = \textbf{while true do skip}$$

Now we will see how to calculate its semantics. We have $\mathscr{C}[\![w]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma_{\textbf{true},\textbf{skip}}$ where

$$\begin{aligned}
\Gamma_{\textbf{true},\textbf{skip}}\varphi\sigma &= \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma), \sigma \\
&= \textbf{true} \to \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma), \sigma \\
&= \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma) \\
&= \varphi^*\sigma \\
&= \varphi\sigma
\end{aligned}$$

So we have $\Gamma_{\textbf{true},\textbf{skip}}\varphi = \varphi$, that is $\Gamma_{\textbf{true},\textbf{skip}}$ is the identity function. Then each function φ is a fixpoint of $\Gamma_{\textbf{true},\textbf{skip}}$, but we are looking for the least fixpoint. This means that the sought solution is the least function in the $CPO_\perp$ of functions $\Sigma \to \Sigma_\perp$. Then we have

$$\text{fix } \Gamma_{\textbf{true},\textbf{skip}} = \lambda\sigma.\, \perp_{\Sigma_\perp}$$

In the following we will often write just Γ when the subscripts b and c are obvious from the context.

Example 6.7. Let us consider the commands

$$w \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } (c\,;\, w) \textbf{ else skip}$$

Now we show the denotational equivalence between w and c' for any b and c. Since $\mathscr{C}[\![w]\!]$ is a fixpoint we have

$$\mathscr{C}[\![w]\!] = \Gamma(\mathscr{C}[\![w]\!]) = \lambda\sigma.\, \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![w]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

For c' we have

$$\begin{aligned}\mathscr{C}[\![\textbf{if } b \textbf{ then } (c;w) \textbf{ else skip}]\!] &= \lambda\sigma.\, \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c;w]\!]\sigma, \mathscr{C}[\![\textbf{skip}]\!]\sigma \\ &= \lambda\sigma.\, \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![w]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma\end{aligned}$$

Hence $\mathscr{C}[\![w]\!] = \mathscr{C}[\![c']\!]$.

Example 6.8. Let us consider the command

$$c \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } x := x - 1$$

we have $\mathscr{C}[\![c]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma$ where[1]

$$\begin{aligned}\Gamma\varphi &\stackrel{\text{def}}{=} \lambda\sigma.\, \mathscr{B}[\![x \neq 0]\!]\sigma \to \varphi^*(\mathscr{C}[\![x := x-1]\!]\sigma), \sigma \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \varphi^*\left(\sigma[{}^{\sigma x - 1}/_x]\right), \sigma \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \varphi\left(\sigma[{}^{\sigma x - 1}/_x]\right), \sigma\end{aligned}$$

Let us compute an approximation of the fixpoint:

$$\begin{aligned}\varphi_0 &= \Gamma^0\, \bot_{\Sigma\to\Sigma_\bot} = \bot_{\Sigma\to\Sigma_\bot} = \lambda\sigma.\, \bot_{\Sigma_\bot} \\ \varphi_1 &= \Gamma\, \varphi_0 \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \underbrace{\bot_{\Sigma\to\Sigma_\bot}}_{\varphi_0}\left(\sigma[{}^{\sigma x - 1}/_x]\right), \sigma \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \bot_{\Sigma_\bot}, \sigma \\ \varphi_2 &= \Gamma\, \varphi_1 \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \underbrace{(\lambda\sigma'.\, \sigma' x \neq 0 \to \bot_{\Sigma_\bot}, \sigma')}_{\varphi_1}\left(\sigma[{}^{\sigma x - 1}/_x]\right), \sigma\end{aligned}$$

Now we have the following possibilities for computing $\varphi_2\sigma$:

[1] Note that in the last step we can remove the lifting operation from φ^* because $\sigma[{}^{\sigma x-1}/_x] \neq \bot$.

$\sigma x < 0$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x \neq 0$ and thus $\varphi_2\sigma = \bot_{\Sigma_\bot}$.
$\sigma x = 0$) Then $\sigma x \neq 0$ is false and thus $\varphi_2\sigma = \sigma = \sigma[^0/_x]$.
$\sigma x = 1$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x = 0$ and thus $\varphi_2\sigma = \sigma[^{\sigma x-1}/_x] = \sigma[^0/_x]$.
$\sigma x > 1$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x \neq 0$ and thus $\varphi_2\sigma = \bot_{\Sigma_\bot}$.

Summarising,

$$\frac{\sigma x < 0}{\varphi_2\sigma = \bot} \qquad \frac{\sigma x = 0}{\varphi_2\sigma = \sigma[^0/_x]} \qquad \frac{\sigma x = 1}{\varphi_2\sigma = \sigma[^0/_x]} \qquad \frac{\sigma x > 1}{\varphi_2\sigma = \bot}$$

So we have

$$\varphi_2 = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < 2 \to \sigma[^0/_x], \bot)$$

We can conjecture that $\forall n \in \mathbb{N}.\ P(n)$, where

$$P(n) \stackrel{\text{def}}{=} (\ \varphi_n = \lambda\sigma.\ 0 \leq \sigma x < n \to \sigma[^0/_x]\ ,\ \bot)$$

We are now ready to prove our conjecture by mathematical induction on n.

Base case: The base case is trivial, indeed we know $\varphi_0 = \lambda\sigma.\ \bot$ and

$$\begin{aligned}\lambda\sigma.\ 0 \leq \sigma x < 0 \to \sigma[^0/_x]\ ,\ \bot &= \lambda\sigma.\,\textbf{false}\ \to \sigma[^0/_x], \bot)\\ &= \lambda\sigma.\ \bot\end{aligned}$$

Inductive case: For the inductive case, let us assume

$$P(n) \stackrel{\text{def}}{=} (\ \varphi_n = \lambda\sigma.\ 0 \leq \sigma x < n \to \sigma[^0/_x]\ ,\ \bot).$$

We want to prove

$$P(n+1) \stackrel{\text{def}}{=} (\ \varphi_{n+1} = \lambda\sigma.\ 0 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot)$$

By definition

$$\varphi_{n+1} = \Gamma\ \varphi_n = \lambda\sigma.\ \sigma x \neq 0 \to \varphi_n(\sigma[^{\sigma x-1}/_x]), \sigma$$

Let $\sigma' \stackrel{\text{def}}{=} \sigma[^{\sigma x-1}/_x]$. By the inductive hypothesis, we have

$$\begin{aligned}\varphi_n(\sigma') &= 0 \leq \sigma' x < n \to \sigma'[^0/_x]\ ,\ \bot\\ &= 0 \leq \sigma x - 1 < n \to \sigma[^0/_x]\ ,\ \bot\\ &= 1 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot\end{aligned}$$

Thus

$$\varphi_{n+1}\sigma = \sigma x \neq 0 \to (\ 1 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot)\ ,\ \sigma$$

Now we have the following possibilities for computing $\varphi_{n+1}\sigma$:

$\sigma x < 0)$ Then $\sigma x \neq 0$ and $\sigma x < 1$ are true, thus $\varphi_{n+1}\sigma = \bot$.

$\sigma x = 0)$ Then $\sigma x \neq 0$ is false and thus $\varphi_{n+1}\sigma = \sigma = \sigma[^0/_x]$.

$1 \leq \sigma x < n+1)$ Then $\sigma x \neq 0$ and $1 \leq \sigma x < n+1$ are true, thus $\varphi_{n+1}\sigma = \sigma[^0/_x]$.

$\sigma x \geq n+1)$ Then $\sigma x \neq 0$ is true, but $1 \leq \sigma x < n+1$ is false, thus $\varphi_{n+1}\sigma = \bot$.

Summarising,

$$\frac{\sigma x < 0}{\varphi_{n+1}\sigma = \bot} \quad \frac{\sigma x = 0}{\varphi_{n+1}\sigma = \sigma[^0/_x]} \quad \frac{1 \leq \sigma x < n+1}{\varphi_{n+1}\sigma = \sigma[^0/_x]} \quad \frac{\sigma x \geq n+1}{\varphi_{n+1}\sigma = \bot}$$

Then

$$\varphi_{n+1} = \lambda\sigma.\ 0 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot$$

which proves $P(n+1)$.

Finally we have

$$\mathscr{C}[\![c]\!] = \text{fix}\ \Gamma = \bigsqcup_{n\in\mathbb{N}} \Gamma^n \bot = \bigsqcup_{n\in\mathbb{N}} \varphi_n = \lambda\sigma.\ 0 \leq \sigma x \to \sigma[^0/_x]\ ,\ \bot$$

6.3 Equivalence Between Operational and Denotational Semantics

This section deals with the issue of equivalence between the two semantics of IMP introduced up to now. As we will show, the denotational and operational semantics agree. As usual we will handle first arithmetic and boolean expressions, then assuming the proved equivalences we will show that the operational and denotational semantics agree also on commands.

6.3.1 Equivalence Proofs for Expressions

We start by considering arithmetic expressions. We want to prove that the operational and denotational semantics coincide, that is, the results of evaluating an arithmetic expression both by operational and denotational semantics are the same. If we regard the operational semantics as an interpreter and the denotational semantics as a compiler we are proving that interpreting an IMP program and executing its compiled version starting from the same memory leads to the same result.

Theorem 6.1. *For all arithmetic expressions $a \in Aexp$, the predicate $P(a)$ holds, where*

$$P(a) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \langle a, \sigma \rangle \to \mathscr{A}[\![a]\!]\sigma$$

Proof. The proof is by structural induction on arithmetic expressions.

Const: $P(n) \stackrel{\text{def}}{=} \forall \sigma.\ \langle n, \sigma \rangle \to \mathscr{A}[\![n]\!]\sigma$ holds because, given a generic σ, we have $\langle n, \sigma \rangle \to n$ and $\mathscr{A}[\![n]\!]\sigma = n$.

Vars: $P(x) \stackrel{\text{def}}{=} \forall \sigma.\ \langle x, \sigma \rangle \to \mathscr{A}[\![x]\!]\sigma$ holds because, given a generic σ, we have $\langle x, \sigma \rangle \to \sigma x$ and $\mathscr{A}[\![x]\!]\sigma = \sigma x$.

Ops: Let us generalise the proof for the binary operations of arithmetic expressions. Consider two arithmetic expressions a_0 and a_1 and a binary operator $\odot \in \{+, -, \times\}$ of IMP, whose corresponding semantic operator is $\cdot$. We assume

$$P(a_0) \stackrel{\text{def}}{=} \langle a_0, \sigma \rangle \to \mathscr{A}[\![a_0]\!]\sigma$$
$$P(a_1) \stackrel{\text{def}}{=} \langle a_1, \sigma \rangle \to \mathscr{A}[\![a_1]\!]\sigma$$

and we want to prove

$$P(a_0 \odot a_1) \stackrel{\text{def}}{=} \langle a_0 \odot a_1, \sigma \rangle \to \mathscr{A}[\![a_0 \odot a_1]\!]\sigma$$

By using the inductive hypothesis we derive

$$\langle a_0 \odot a_1, \sigma \rangle \to \mathscr{A}[\![a_0]\!]\sigma \cdot \mathscr{A}[\![a_1]\!]\sigma$$

Finally, by definition of $\mathscr{A}$

$$\mathscr{A}[\![a_0]\!]\sigma \cdot \mathscr{A}[\![a_1]\!]\sigma = \mathscr{A}[\![a_0 \odot a_1]\!]\sigma$$

□

The case of boolean expressions is completely similar to that of arithmetic expressions, so we leave the proof as an exercise (see Problem 6.2).

Theorem 6.2. *For all boolean expressions $b \in Bexp$, the predicate $P(b)$ holds, where*

$$P(b) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \langle b, \sigma \rangle \to \mathscr{B}[\![b]\!]\sigma$$

From now on we will assume the equivalence between denotational and operational semantics for boolean and arithmetic expressions.

6.3.2 Equivalence Proof for Commands

Central to the proof of equivalence between denotational and operational semantics is the case of commands. Operational and denotational semantics are defined in

very different formalisms: on the one hand we have an inference rule system which allows us to calculate the execution of each command; on the other hand we have a function which associates with each command its functional meaning. So to show the equivalence between the two semantics we will prove the following property.

Theorem 6.3. $\forall c \in Com.\ \forall \sigma, \sigma' \in \Sigma. \quad \langle c, \sigma \rangle \to \sigma' \quad \Leftrightarrow \quad \mathscr{C}[\![c]\!]\sigma = \sigma'.$

As usual we divide the proof into two parts:

Correctness: $\forall c \in Com,\ \forall \sigma, \sigma' \in \Sigma$ we prove

$$P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$$

Completeness: $\forall c \in Com$ we prove

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \in \Sigma. \quad \mathscr{C}[\![c]\!]\sigma = \sigma' \quad \Rightarrow \quad \langle c, \sigma \rangle \to \sigma'$$

Notice that in this way the undefined cases are also handled for the equivalence: for instance we have as a corollary that

$$\langle c, \sigma \rangle \not\to \quad \Rightarrow \quad \mathscr{C}[\![c]\!]\sigma = \perp_{\Sigma_\perp}$$

since otherwise, assuming $\mathscr{C}[\![c]\!]\sigma = \sigma'$ for some $\sigma' \in \Sigma$, it would follow that $\langle c, \sigma \rangle \to \sigma'$. Similarly in the opposite direction:

$$\mathscr{C}[\![c]\!]\sigma = \perp_{\Sigma_\perp} \quad \Rightarrow \quad \langle c, \sigma \rangle \not\to$$

6.3.2.1 Correctness

Let us prove the first part of Theorem 6.3. We let

$$P\left(\langle c, \sigma \rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$$

and prove that $P(\langle c, \sigma \rangle \to \sigma')$ holds for any $c \in Com$ and $\sigma, \sigma' \in \Sigma$.

We proceed by rule induction. So for each rule we will assume the property holds for the premises and we will prove that the property holds for the conclusion.

skip: Let us consider the operational rule for the **skip**:

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma}$$

We want to prove

$$P(\langle \mathbf{skip}, \sigma \rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{skip}]\!]\sigma = \sigma$$

Obviously the proposition is true by the definition of the denotational semantics.

assign: Let us consider the rule for the assignment command:

$$\frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma\,[^{m}/_{x}]}$$

We assume $\langle a, \sigma\rangle \to m$ and hence $\mathscr{A}[\![a]\!]\,\sigma = m$ by the equivalence of the operational and denotational semantics of arithmetic expressions.
We want to prove

$$P(\langle x := a, \sigma\rangle \to \sigma\,[^{m}/_{x}]) \stackrel{\text{def}}{=} \mathscr{C}[\![x := a]\!]\,\sigma = \sigma\,[^{m}/_{x}]$$

By the definition of the denotational semantics

$$\mathscr{C}[\![x := a]\!]\,\sigma = \sigma[^{\mathscr{A}[\![a]\!]\sigma}/_{x}] = \sigma[^{m}/_{x}]$$

seq: Let us consider the concatenation rule:

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}$$

We assume

$$P(\langle c_0, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0]\!]\,\sigma = \sigma''$$
$$P(\langle c_1, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_1]\!]\,\sigma'' = \sigma'$$

We want to prove

$$P(\langle c_0; c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0; c_1]\!]\,\sigma = \sigma'$$

By the denotational semantics definition and the inductive hypotheses

$$\mathscr{C}[\![c_0; c_1]\!]\,\sigma = \mathscr{C}[\![c_1]\!]^{*}\,(\mathscr{C}[\![c_0]\!]\,\sigma) = \mathscr{C}[\![c_1]\!]^{*}\,\sigma'' = \mathscr{C}[\![c_1]\!]\,\sigma'' = \sigma'$$

Note that the lifting operator can be removed because $\sigma'' \neq \perp$ by the inductive hypothesis.

iftt: Let us consider the rule

$$\frac{\langle b, \sigma\rangle \to \mathbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to \sigma'}$$

We assume

- $\langle b, \sigma\rangle \to \mathbf{true}$ and therefore $\mathscr{B}[\![b]\!]\,\sigma = \mathbf{true}$ by the correspondence between the operational and denotational semantics for boolean expressions;
- $P(\langle c_0, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0]\!]\,\sigma = \sigma'$

We want to prove

$$P(\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$$

In fact, we have

$$\begin{aligned}\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \\ &= \textbf{true} \to \sigma', \mathscr{C}[\![c_1]\!]\sigma \\ &= \sigma'\end{aligned}$$

ifff: The proof for the second rule of the conditional command is completely analogous to the previous one and thus omitted.

whff: Let us consider the rule

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$

We assume $\langle b, \sigma\rangle \to \textbf{false}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$.
We want to prove

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma$$

By the fixpoint property of the denotational semantics

$$\begin{aligned}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \textbf{false} \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \sigma\end{aligned}$$

whtt: At last we consider the second rule of the while command:

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

We assume

- $\langle b, \sigma\rangle \to \textbf{true}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$
- $P(\langle c, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma''$
- $P(\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' = \sigma'$

We want to prove

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma'$$

By the definition of the denotational semantics and the inductive hypotheses

$$\begin{aligned}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \textbf{true} \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*\sigma'', \sigma \\ &= \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*\sigma'' \\ &= \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' \\ &= \sigma'\end{aligned}$$

Note that the lifting operator can be removed since $\sigma'' \neq \bot$.

6.3.2.2 Completeness

Let us conclude the proof of Theorem 6.3 by showing that, for all $c \in Com$

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \in \Sigma.\ \mathscr{C}[\![c]\!]\sigma = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma'$$

Since the denotational semantics is given by structural recursion we will proceed by induction on the structure of commands.

skip: We need to prove

$$P(\textbf{skip}) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\textbf{skip}]\!]\sigma = \sigma' \Rightarrow \langle \textbf{skip}, \sigma\rangle \to \sigma'$$

By definition we have $\mathscr{C}[\![\textbf{skip}]\!]\sigma = \sigma$ and $\langle \textbf{skip}, \sigma\rangle \to \sigma$ is an axiom of the operational semantics.

assign: We need to prove

$$P(x := a) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![x := a]\!]\sigma = \sigma' \Rightarrow \langle x := a, \sigma\rangle \to \sigma'$$

By the denotational semantics definition we have $\sigma' = \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}]$ and by the equivalence between operational and denotational semantics for expressions we have $\langle a, \sigma\rangle \to \mathscr{A}[\![a]\!]\sigma$, thus we can apply the rule (assign) to conclude

$$\langle x := a, \sigma\rangle \to \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}]$$

seq: We assume

- $P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma''.\ \mathscr{C}[\![c_0]\!]\sigma = \sigma'' \Rightarrow \langle c_0, \sigma\rangle \to \sigma''$
- $P(c_1) \stackrel{\text{def}}{=} \forall \sigma'', \sigma'.\ \mathscr{C}[\![c_1]\!]\sigma'' = \sigma' \Rightarrow \langle c_1, \sigma''\rangle \to \sigma'$

We want to prove

$$P(c_0; c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![c_0; c_1]\!]\sigma = \sigma' \Rightarrow \langle c_0; c_1, \sigma\rangle \to \sigma'$$

Let us assume $\mathscr{C}[\![c_0; c_1]\!]\sigma = \sigma'$, the premise of the implication, and prove the conclusion $\langle c_0; c_1, \sigma\rangle \to \sigma'$. We have

$$\mathscr{C}[\![c_0;c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) = \sigma'$$

Since $\sigma' \neq \perp$, it must be that $\mathscr{C}[\![c_0]\!]\sigma \neq \perp$, i.e., we can assume the termination of c_0 and thus omit the lifting operator:

$$\mathscr{C}[\![c_0;c_1]\!]\sigma = \mathscr{C}[\![c_1]\!](\mathscr{C}[\![c_0]\!]\sigma) = \sigma'$$

Let $\mathscr{C}[\![c_0]\!]\sigma = \sigma''$. We have $\mathscr{C}[\![c_1]\!]\sigma'' = \sigma'$. Then we can apply modus ponens to the inductive assumptions $P(c_0)$ and $P(c_1)$, to get $\langle c_0, \sigma\rangle \to \sigma''$ and $\langle c_1, \sigma''\rangle \to \sigma'$. Thus we can apply the inference rule:

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \qquad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}$$

to conclude $\langle c_0; c_1, \sigma\rangle \to \sigma'$.

if: We assume

- $P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![c_0]\!]\sigma = \sigma' \Rightarrow \langle c_0, \sigma\rangle \to \sigma'$
- $P(c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![c_1]\!]\sigma = \sigma' \Rightarrow \langle c_1, \sigma\rangle \to \sigma'$

We need to prove

$$P(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma' \Rightarrow \langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$$

Let us assume the premise $\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$ and prove the conclusion $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$. By definition

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma$$

Now, either $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ or $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have also $\langle b, \sigma\rangle \to \textbf{false}$. Then

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]\sigma = \sigma'$$

By modus ponens on the inductive hypothesis $P(c_1)$ we have $\langle c_1, \sigma\rangle \to \sigma'$. Thus we can apply the rule

$$\frac{\langle b, \sigma\rangle \to \textbf{false} \qquad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$

to conclude $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$.
The case where $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ is completely analogous and thus omitted.

while: We assume

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma''.\ \mathscr{C}[\![c]\!]\sigma = \sigma'' \Rightarrow \langle c, \sigma\rangle \to \sigma''$$

We need to prove

$$P(\textbf{while } b \textbf{ do } c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

By definition $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \text{fix } \Gamma_{b,c}\,\sigma = \left(\bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}\bot\right)\sigma$ so

$$\begin{gathered}
\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\
\Leftrightarrow \\
\left(\bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}\bot\right)\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\
\Leftrightarrow \\
\left(\exists n \in \mathbb{N}.\ (\Gamma^n_{b,c}\bot)\sigma = \sigma'\right) \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\
\Leftrightarrow \\
\forall n \in \mathbb{N}.\ \left(\Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'\right)
\end{gathered}$$

Let $A(n) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$.
We prove that $\forall n \in \mathbb{N}.\ A(n)$ by mathematical induction.

Base case: We have to prove $A(0)$, namely

$$\forall \sigma, \sigma'.\ \Gamma^0_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

Since $\Gamma^0_{b,c}\bot\sigma = \bot\sigma = \bot$ and $\sigma' \neq \bot$ the premise is false and hence the implication is true.

Ind. case: Let us assume

$$A(n) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

We want to show that

$$A(n+1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \Gamma^{n+1}_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

We assume $\Gamma^{n+1}_{b,c}\bot\sigma = \Gamma_{b,c}\left(\Gamma^n_{b,c}\bot\right)\sigma = \sigma'$, that is

$$\mathscr{B}[\![b]\!]\sigma \to \left(\Gamma^n_{b,c}\bot\right)^*\left(\mathscr{C}[\![c]\!]\sigma\right), \sigma = \sigma'$$

Now either $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ or $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.

- If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have $\langle b, \sigma\rangle \to \textbf{false}$ and $\sigma' = \sigma$. Now by using the rule (whff)

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$

we conclude $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma$.

- if $\mathscr{B}[\![b]\!]\,\sigma = \mathbf{true}$ we have $\langle b,\sigma\rangle \to \mathbf{true}$ and

$$\left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}[\![c]\!]\,\sigma) = \sigma'$$

Since $\sigma' \neq \bot$ there must exist some $\sigma'' \neq \bot$ with $\mathscr{C}[\![c]\!]\,\sigma = \sigma''$ and by structural induction $\langle c,\sigma\rangle \to \sigma''$. Since $\left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}[\![c]\!]\,\sigma) = \left(\Gamma^n_{b,c}\bot\right)\sigma'' = \sigma'$ we have by the mathematical induction hypothesis $A(n)$ that

$$\left\langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma''\right\rangle \to \sigma'$$

Finally, by using the rule (whtt)

$$\frac{\langle b,\sigma\rangle \to \mathbf{true} \quad \langle c,\sigma\rangle \to \sigma'' \quad \left\langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma''\right\rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma\rangle \to \sigma'}$$

we conclude $\langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma\rangle \to \sigma'$.

6.4 Computational Induction

How are we able to prove properties about fixpoints? To fill this gap we introduce Scott's *computational induction*, which applies to a class of properties corresponding to inclusive sets.

Definition 6.10 (Inclusive property). Let $(D,\sqsubseteq)$ be a CPO, let $P \subseteq D$ be a set. We say that P is an *inclusive* set if and only if

$$(\forall n \in \mathbb{N}.\ d_n \in P) \Rightarrow \bigsqcup_{n\in\mathbb{N}} d_n \in P$$

A property is *inclusive* if the set of values on which it holds is inclusive.

Intuitively, a set P is inclusive if whenever we form a chain out of elements in P, then the limit of the chain is also in P, i.e., P is inclusive if and only if it forms a CPO.

Example 6.9 (Non-inclusive property). Let $(\{a,b\}^* \cup \{a,b\}^\infty, \sqsubseteq)$ be a CPO where $x \sqsubseteq y \Leftrightarrow \exists z.\ y = xz$. So the elements of the CPO are sequences of a and b and $x \sqsubseteq y$ iff x is a finite prefix of y. Let us now define the following property:

- $x \in \{a,b\}^* \cup \{a,b\}^\infty$ is fair iff $\nexists y \in \{a,b\}^*.\ x = ya^\infty \ \vee\ x = yb^\infty$

Fairness is the property of an arbiter which does not favour one of two competitors all the time from some point on. Fairness is not inclusive, indeed,

- the sequence a^n is finite and thus fair for any $n \in \mathbb{N}$;

- $\bigsqcup_{n\in\mathbb{N}} a^n = a^\infty$;
- a^∞ is obviously not fair.

Theorem 6.4 (Computational induction). *Let P be a property, $(D,\sqsubseteq)$ a CPO$_\perp$ and f a monotone and continuous function on it. Then the inference rule*

$$\frac{P \text{ inclusive} \quad \bot \in P \quad \forall d \in D.\ (d \in P \Rightarrow f(d) \in P)}{\mathrm{fix}\, f \in P}$$

is sound.

Proof. Given the second and third premises, it is easy to prove by mathematical induction that $\forall n.\ f^n(\bot) \in P$. Then also $\bigsqcup_{n\in\mathbb{N}} f^n(\bot) \in P$ since P is inclusive, and $\mathrm{fix}(f) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$. □

Example 6.10 (Computational induction). Let us consider the command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } x := x - 1$$

from Example 6.8. We want to prove the property

$$\mathscr{C}[\![\textbf{while } x \neq 0 \textbf{ do } x := x-1]\!]\,\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$$

By definition

$$\mathscr{C}[\![w]\!] = \mathrm{fix}\ \Gamma \quad \text{where} \quad \Gamma \stackrel{\text{def}}{=} \lambda\varphi.\ \lambda\sigma.\ \sigma x \neq 0 \rightarrow \varphi\sigma[^{\sigma x - 1}/_x], \sigma$$

Let us define the property

$$P(\varphi) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ (\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x])$$

We will show that the property is inclusive, that is, given a chain $\{\varphi_i\}_{i\in\mathbb{N}}$ we have

$$(\forall i \in \mathbb{N}.\ P(\varphi_i)) \Rightarrow P(\bigsqcup_{i\in\mathbb{N}} \varphi_i)$$

Let us assume $\forall i \in \mathbb{N}.\ P(\varphi_i)$, namely that

$$\forall i, \sigma, \sigma'.\ \left(\varphi_i\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\right)$$

We want to prove that

$$\forall \sigma, \sigma'. \left((\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x] \right)$$

Suppose $(\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma'$. We are left to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$. By $(\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma'$ we have that $\exists k \in \mathbb{N}.\ \varphi_k\sigma = \sigma'$. Then we can conclude the thesis by $P(\varphi_k)$.

We can now apply the computational induction

$$\frac{P \text{ inclusive} \quad P(\bot) \quad \forall \varphi.\, P(\varphi) \Rightarrow P(\Gamma\varphi)}{P(\text{fix } \Gamma)}$$

as $P(\text{fix } \Gamma) = P(\mathscr{C}[\![w]\!])$.

P inclusive: It has been proved above.

$P(\bot)$: It is obvious, since $\bot\sigma = \sigma'$ is always false.

$\forall \varphi.\, P(\varphi) \Rightarrow P(\Gamma\varphi)$: We assume

$$P(\varphi) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\, (\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}])$$

and we want to prove

$$P(\Gamma\varphi) = \forall \sigma, \sigma'.\, (\Gamma\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}])$$

We assume the premise

$$\Gamma\varphi\sigma = \big(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x - 1}/_{x}], \sigma\big) = \sigma'$$

and need to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}]$. There are two cases to consider:

- If $\sigma x = 0$, we have

  $$(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x - 1}/_{x}], \sigma) = \sigma$$

  therefore $\sigma' = \sigma$ and trivially

  $$\sigma x = 0 \geq 0 \qquad \sigma' = \sigma = \sigma[{}^{0}/_{x}]$$

- If $\sigma x \neq 0$, we have

  $$(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x - 1}/_{x}], \sigma) = \varphi\sigma[{}^{\sigma x - 1}/_{x}]$$

  Let $\sigma'' = \sigma[{}^{\sigma x - 1}/_{x}]$. We exploit $P(\varphi)$ over σ'', σ':

  $$\varphi \underbrace{\sigma[{}^{\sigma x - 1}/_{x}]}_{\sigma''} = \sigma' \Rightarrow \sigma'' x \geq 0 \wedge \sigma' = \sigma''[{}^{0}/_{x}]$$

  We have

  $$\sigma'' x \geq 0 \Leftrightarrow \sigma[{}^{\sigma x - 1}/_{x}]x \geq 0 \Leftrightarrow \sigma x \geq 1 \Rightarrow \sigma x \geq 0$$

  $$\sigma' = \sigma''[{}^{0}/_{x}] = \sigma[{}^{\sigma x - 1}/_{x}][{}^{0}/_{x}] = \sigma[{}^{0}/_{x}]$$

Finally, we can conclude by computational induction that the property P holds for the fixpoint fix Γ and thus for the semantics of the command w as $\mathscr{C}[\![w]\!] = \text{fix } \Gamma$.

Problems

6.1. The following problems serve to get acquainted with the use of variables in the lambda notation.

1. Is $\lambda x.\ \lambda x.\ x$ α-convertible to one or more of the following expressions?

   a. $\lambda y.\ \lambda x.\ x$
   b. $\lambda y.\ \lambda x.\ y$
   c. $\lambda y.\ \lambda y.\ y$
   d. $\lambda x.\ \lambda y.\ x$
   e. $\lambda z.\ \lambda w.\ w$

2. Is $((\lambda x.\ \lambda y.\ x)\ y)$ equivalent to one or more of the following expressions?

   a. $\lambda y.\ \lambda y.\ y$
   b. $\lambda y.\ y$
   c. $\lambda y.\ z$
   d. $\lambda z.\ y$
   e. $\lambda x.\ y$

6.2. Prove Theorem 6.2.

6.3. Prove that the commands

$$c \stackrel{\text{def}}{=} x := 0;\ \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2 \qquad\qquad c' \stackrel{\text{def}}{=} x := 0;\ c_1$$

are semantically equivalent for any commands c_1, c_2. Carry out the proof using both the operational semantics and the denotational semantics.

6.4. Prove that the two commands

$$w \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c \qquad\qquad w' \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } (\textbf{if } b \textbf{ then } c \textbf{ else skip})$$

are equivalent for any b and c using the denotational semantics.

6.5. Prove that $\mathscr{C}[\![\textbf{while true do skip}]\!] = \mathscr{C}[\![\textbf{while true do } x := x+1]\!]$.

6.6. Prove that $\mathscr{C}[\![\textbf{while } x \neq 0 \textbf{ do } x := 0]\!] = \mathscr{C}[\![x := 0]\!]$.

6.7. Prove that

$$\mathscr{C}[\![\textbf{while } x = 0 \textbf{ do skip}]\!] = \mathscr{C}[\![\textbf{if } x = 0 \textbf{ then } (\textbf{while true do } x := 0) \textbf{ else skip}]\!].$$

6.8. Add to **IMP** the command

$$\textbf{repeat } n \textbf{ times } c$$

with n a natural number. Its denotational semantics is

$$\mathscr{C}[\![\textbf{repeat } n \textbf{ times } c]\!]\sigma = (\mathscr{C}[\![c]\!])^n\sigma$$

1. Define the operational semantics of the repeat command.
2. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.
3. Prove that, when the "while" construct is replaced by the "repeat" construct in the syntax of IMP, then the execution of every command terminates.

6.9. Add to IMP the command

$$\textbf{reset } x \textbf{ in } c$$

with the following informal meaning: execute the command c in the state where x is reset to 0, then after the execution of c reassign to location x its original value.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.10. Add to **IMP** the command

$$\textbf{do } c \textbf{ undoif } b$$

with the following informal meaning: execute c; if after the execution of c the boolean expression b is satisfied, then go back to the state before the execution of c.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.11. Extend IMP with the command

$$\textbf{try } c_1 = c_2 \textbf{ else } c_3$$

that returns the store obtained by computing c_1 if it coincides with the one obtained by computing c_2; if they differ it returns the store obtained by computing c_3; it diverges otherwise.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.

3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.12. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do } (r := r \times x \,;\, y := y - 1)$$

Compute the denotational semantics $\mathscr{C}[\![w]\!] = \text{fix}\, \Gamma$.

Hint: Prove that letting $\varphi_n \stackrel{\text{def}}{=} \Gamma^n \bot_{\Sigma \to \Sigma_\bot}$ it holds that $\forall n \geq 1$

$$\varphi_n = \lambda \sigma.\, (\sigma y > 0) \to (\, (\sigma y \geq n) \to \bot_{\Sigma_\bot} \,,\, \sigma[{}^{\sigma r \times (\sigma x)^{\sigma y}}/_r, {}^{0}/_y] \,) \,,\, \sigma.$$

6.13. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } (x := x - 1 \,;\, y := y + 1)$$

Prove, using Scott's computational induction, that for all σ, σ' we have

$$\mathscr{C}[\![w]\!]\, \sigma = \sigma' \quad \Rightarrow \quad \sigma(x) \geq 0 \wedge \sigma' = \sigma[{}^{\sigma(x)+\sigma(y)}/_y, {}^{0}/_x]$$

Part III
HOFL: a Higher-Order Functional Language

This part focuses on models for sequential computations that are associated with HOFL, a higher-order declarative language that follows the functional style. Chapter 7 presents the syntax, typing and operational semantics of HOFL, while Chapter 9 defines its denotational semantics. The two are related in Chapter 10. Chapter 8 extends the theory presented in Chapter 5 to allow the definition of more complex domains, as needed by the type constructors available in HOFL.

Chapter 7
Operational Semantics of HOFL

Typing is no substitute for thinking. (Richard Hamming)

Abstract In the previous part of the book we have introduced and studied an imperative language called IMP. In this chapter we move our attention to functional languages. In particular, we introduce HOFL, a simple higher-order functional language that allows for the explicit construction of infinitely many types. We overview Church and Curry type theories. Then, we present a *lazy* operational semantics, which corresponds to a *call-by-name* strategy, namely actual parameters are passed to functions without evaluating them. This view is contrasted with the *eager* evaluation semantics, which corresponds to a *call-by-value* strategy, where all actual parameters are evaluated before being passed to functions. The operational semantics evaluates (well-typed) terms to suitable canonical forms.

7.1 Syntax of HOFL

We start by introducing the plain syntax of HOFL. Then we discuss the type theory and define the well-formed terms. Finally we present the operational semantics of well-formed terms, which reduces terms to their canonical form (when they exist).

In IMP there are only three types: *Aexp* for arithmetic expressions, *Bexp* for boolean expressions and *Com* for commands. Since IMP does not allow explicit construction of other types, these types are directly embedded in its syntax. HOFL, instead, allows one to define a variety of types, so we first present the grammar for *pre-terms*, then we introduce the concept of typed terms, namely the well-formed sentences of HOFL. Due to the context-sensitive constraints induced by the types, it is possible to see that well-formed terms could not be defined by a syntax expressed in a context-free format. We assume a set *Var* of variables is given.

Definition 7.1 (HOFL syntax). The following productions define the syntax of HOFL pre-terms:

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_7

$$t \quad ::= \quad x \mid n \mid t_0 + t_1 \mid t_0 - t_1 \mid t_0 \times t_1 \mid \textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \mid (t_0, t_1) \mid \textbf{fst}(t) \mid \textbf{snd}(t) \mid \lambda x.\, t \mid (t_0\, t_1) \mid \textbf{rec}\, x.\, t$$

where x is a variable and n an integer.

Besides usual variables x, constants n and arithmetic operators $+, -, \times$, we find a conditional construct **if** t **then** t_0 **else** t_1 that reads as **if** $t = 0$ **then** t_0 **else** t_1; the constructs for pairing terms (t_0, t_1) and for projecting over the first and second component of a pair $\textbf{fst}(t)$ and $\textbf{snd}(t)$; function abstraction $\lambda x.\, t$ and application $(t_0\, t_1)$; and recursive definition $\textbf{rec}\, x.\, t$. Recursion allows us to define recursive terms, namely $\textbf{rec}\, x.\, t$ defines a term t that can contain variable x, which in turn can be replaced by its recursive definition $\textbf{rec}\, x.\, t$.

We call *pre-terms* the terms generated by the syntax above, because it is evident that one could write ill-formed terms, for example by applying a projection to an integer instead of a pair ($\textbf{fst}(1)$) or summing an integer to a function ($1 + \lambda x.\, x$). To avoid these constructions we introduce the concepts of *type* and *typed term*.

7.1.1 Typed Terms

Definition 7.2 (HOFL types). A HOFL type is a term constructed by using the following grammar:

$$\tau \quad ::= \quad int \quad \mid \quad \tau_0 * \tau_1 \quad \mid \quad \tau_0 \to \tau_1$$

We let $\mathscr{T}$ denote the set of all types.

We allow constant type int, the pair type $\tau_0 * \tau_1$ and the function type $\tau_0 \to \tau_1$. Using these productions we can define infinitely many types, such as $(int * int) \to int$ for functions that take as argument a pair of integers and return an integer, and $int \to (int * (int \to int))$ for functions that take an integer and return an integer paired with a function from integers to integers.

Now we define the rule system which allows us to say whether a pre-term of HOFL is well-formed (i.e., whether we can associate a type expressed in the above grammar with a given pre-term). The predicates we are interested in are of the form $t : \tau$, expressing that the pre-term t is well-formed and has type τ. We assume variables are typed, i.e., that a function $\widehat{(\cdot)} : Var \to \mathscr{T}$ is given, which assigns a unique type to each variable.

$$\frac{}{x : \widehat{x}}$$

The rule for variables assigns to each variable x its type $\widehat{x}$.

$$\frac{}{n : int} \qquad \frac{t_0 : int \quad t_1 : int}{t_0 \text{ op } t_1 : int}\ \text{op} \in \{+, -, \times\} \qquad \frac{t : int \quad t_0 : \tau \quad t_1 : \tau}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 : \tau}$$

The rules for arithmetic expressions assign type *int* to each integer n and to each expression built using $+, -, \times$, whose arguments must be of type *int* too. The rule for conditional expressions **if** t **then** t_0 **else** $t_1 : \tau$ requires the condition t to be of type *int* and the two branches t_0 and t_1 to have the same type τ, which is also the type of the conditional expression.

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

The rule for pairing says that the type of a term (t_0, t_1) is the pair type $\tau_0 * \tau_1$, where t_i has type τ_i for $i = 0, 1$. Vice versa, for projections it is required that the argument t has pair type $\tau_0 * \tau_1$ for some τ_0 and τ_1, and the result has type τ_0 when the first projection is used or τ_1 when the second projection is used.

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.\, t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

The rule for function abstraction assigns to $\lambda x.\, t$ the functional type $\tau_0 \to \tau_1$, where τ_0 is the type of x and τ_1 is the type of t. In the case of function application $(t_1\ t_0)$, it is required that t_1 has functional type $\tau_0 \to \tau_1$ for some types τ_0 and τ_1, where τ_0 is also the type of t_0. Then, the result has type τ_1.

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.\, t : \tau}$$

The last rule handles recursion: it check that the type τ of the defining expression t is the same as the type of the recursively defined name x; if so, then τ is also the type of the recursive expression **rec** $x.\, t$.

Definition 7.3 (Well-formed terms). Let t be a pre-term of HOFL. We say that t is *well-formed* (or *well-typed*, or *typable*) if there exists a type τ such that $t : \tau$.

We name T_τ the set of well-formed terms of type τ.

Note that our type system is very simple. Indeed it does not allow us to construct useful types, such as recursive, parametric, dependent, polymorphic or abstract types. These limitations imply that many useful terms are discarded. For instance, while it is easy to express the types for lists of integer numbers of fixed length (using the type pairing operator $*$) and functions that manipulate them, in our type system lists of integers of variable length are not typable, because some form of recursion is needed at the level of types to express them.

7.1.2 Typability and Typechecking

As we said in the last section we will give semantics only to well-formed terms, namely terms which have a type in our type system. Therefore we need an algorithm to decide whether a term is well-formed. In this section we will present two different solutions to the typability problem, introduced by Church and by Curry, respectively.

7.1.2.1 Church Type Theory

In Church type theory we explicitly associate a type with each variable and deduce the type of each term by structural recursion (i.e., by using the inference rules in a bottom-up fashion).

In this case, we sometimes directly annotate the bounded variables with their type, for example $\lambda x : int.\ x+x$ or $\mathbf{rec}\ f : int \to int.\ \lambda x : int.\ fx$.

Example 7.1 (Factorial with Church types). Let $x : int$ and $f : int \to int$ in the pre-term

$$fact \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ (x \times (f(x-1)))$$

So we can type *fact* and all its subterms as below:

$$\dfrac{\dfrac{\widehat{f} = int \to int}{f : int \to int} \qquad \dfrac{\dfrac{\widehat{x} = int}{x : int} \qquad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int} \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{\dfrac{\widehat{f} = int \to int}{f : int \to int} \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{x - 1 : int}}{f(x-1) : int}}{(x \times (f(x-1))) : int}}{\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ (x \times (f(x-1))) : int}}{\lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ (x \times (f(x-1))) : int \to int}}{fact : int \to int}$$

More concisely, we write

$$fact \stackrel{\text{def}}{=} \mathbf{rec}\ \underbrace{f}_{int\to int} .\ \underbrace{\lambda \underbrace{x}_{int} .\ \underbrace{\mathbf{if}\ \underbrace{x}_{int}\ \mathbf{then}\ \underbrace{1}_{int}\ \mathbf{else}\ \underbrace{\underbrace{x}_{int} \times (\underbrace{\underbrace{f}_{int\to int}\ (\underbrace{\underbrace{x}_{int} - \underbrace{1}_{int}}_{int})}_{int})}_{int}}_{int})}_{int \to int} \qquad : int \to int$$

7.1.2.2 Curry Type Theory

In Curry style, we do not need to explicitly declare the type of each variable. Instead we use the inference rules to calculate type equations (i.e., equations which have types as variables) whose solutions define all the possible type assignments for the term. This means that the result will be a set of types associated with the typed term. The surprising fact is that this set can be represented as all the instances of a single type term with variables, where one instance is obtained by freely replacing each variable with any type. We call this term with variables the *principal type* of the term. This construction is made by using the rules in a goal-oriented fashion, as we have done in Example 7.5.

Example 7.2 (Identity). Let us consider the identity function

$$\lambda x.\, x$$

By using the type system we have

$$\lambda x.\, x : \tau \quad \nwarrow_{\tau=\tau_1\to\tau_2,\ \widehat{x}=\tau_1} \quad x : \tau_2$$
$$\nwarrow_{\widehat{x}=\tau_2} \quad \square$$

So we have $\widehat{x} = \tau_1 = \tau_2$ and the principal type of $\lambda x.\, x$ is $\tau_1 \to \tau_1$. Now each solution of the type equation will be an identity function for a specified type. For example if we set $\tau_1 = int$ we have $\tau = int \to int$, but if we set $\tau_1 = int * (int \to int)$ we have $\tau = (int * (int \to int)) \to (int * (int \to int))$.

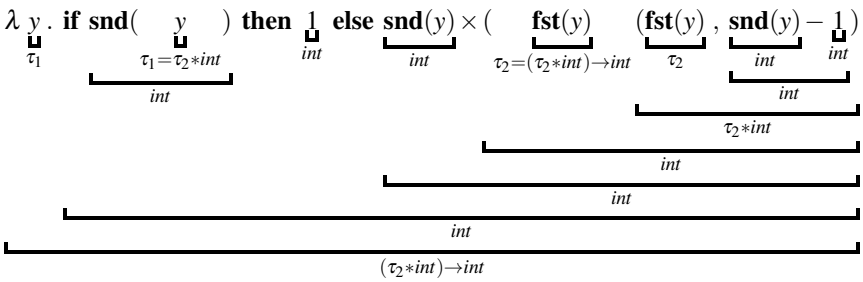
Example 7.3 (Non-typable term of HOFL). Let us consider the following function, which computes the factorial without using recursion.

begin
$fact(f,x) \stackrel{\text{def}}{=}$ **if** $x = 0$ **then** 1 **else** $x \times f(f, x-1)$
$fact(fact, 3)$
end

The first instruction defines *fact* as a function that takes two arguments (e.g., a function f and an integer x) and returns 1 if $x = 0$ and returns $x \times f(f, x-1)$ otherwise. The second instruction invokes *fact* by passing *fact* as a first argument and the number 3 as second argument. Since $3 \neq 0$, the invocation will trigger the calculation $3 \times fact(fact, 2)$ and so on. It can be translated to HOFL as follows:

$$fact \stackrel{\text{def}}{=} \lambda y.\ \textbf{if}\ \textbf{snd}(y) = 0\ \textbf{then}\ 1\ \textbf{else}\ \textbf{snd}(y) \times \textbf{fst}(y)(\textbf{fst}(y), \textbf{snd}(y) - 1)$$

We can try to infer the type of *fact* as follows:



We derive $\mathbf{fst}(y) : \tau_2$ and $\mathbf{fst}(y) : (\tau_2 * int) \to int$. Thus we have $\tau_2 = (\tau_2 * int) \to int$ which has no solution.

We recall the unification algorithm from Section 2.1.4, which can be used to solve general systems of type equations as well. We recall it here, in compact form, to explicitly address the unification of terms that denote types. The idea is that types are terms built over a suitable signature. In the case of HOFL, the signature just consists of the constant *int* and two binary operators $*$ and $\to$, and variables are usually denoted as τs. We start from a system of type equations of the form

$$\begin{cases} t_1 = t_1' \\ t_2 = t_2' \\ \dots \\ t_k = t_k' \end{cases}$$

and then we apply iteratively in any order the following steps:

1. We eliminate all the equations of the form $\tau = \tau$ for τ a type variable.
2. For each equation of the form $f(u_1, \dots, u_n) = f'(u_1', \dots, u_m')$[1]

   if $f \neq f'$: then the system has no solutions and we stop.
   if $f = f'$: then $n = m$ so we must have

   $$u_1 = u_1', u_2 = u_2', \dots, u_n = u_n'$$

   and thus we replace the original equation with these.

3. For each equation of the form $\tau = t$ with $t \neq \tau$:

   if τ appears in t: then the system has no solutions.
   if τ does not appear in t: we replace each occurrence of τ with t in all the other equations.

Eventually, either the system is recognised as unsolvable, or all the variables in the original equations are assigned to solution terms. Note that the order of the step executions can affect the complexity of the algorithm but not the solution. The best

[1] In our case f and f' can be taken from $\{int, *, \to\}$.

execution strategies yield a complexity linear or quasi-linear in the size of the original system of equations.

Example 7.4. Let us now apply the algorithm to the Example 7.3. We have the type equation

$$\tau_2 = (\tau_2 * int) \to int$$

1. We cannot apply step 1 of the algorithm, because the equation does not express a trivial equality.
2. We cannot apply step 2 either, because the left-hand side of the equation consists of a variable and not of an operator applied to some subterms, as required.
3. Step 3 can be applied and it fails, because the type variable τ_2 appears in the right-hand side.

Here we show another interesting term which is not typable.

Example 7.5 (Non-typable terms). Let us define a pre-term t which, when applied to the argument 0, should define the list of all even numbers:

$$t \stackrel{\text{def}}{=} \textbf{rec}\ p.\ \lambda x.\ (x, (p\ (x+2)))$$

Intuitively, the application $t\ 0$ takes the value 0 and places it in the first position of a pair whose second component is the term t itself applied to $0+2=2$, so recursively $t\ 0$ should represent the infinite list of all even numbers:

$$t\ 0 \equiv (0, (t\ 2)) \equiv (0, (2, (t\ 4))) \equiv \cdots \equiv (0, (2, (4, \ldots)))$$

Let us show that this term is not typable:

$$\begin{array}{rl}
t = \textbf{rec}\ p.\ \lambda x.\ (x, (p\ (x+2))) : \tau & \nwarrow_{\widehat{p}=\tau}\ \lambda x.\ (x, (p\ (x+2))) : \tau \\
& \nwarrow_{\tau=\tau_1\to\tau_2,\ \widehat{x}=\tau_1}\ (x, (p\ (x+2))) : \tau_2 \\
& \nwarrow_{\tau_2=\tau_3*\tau_4}\ x : \tau_3,\ (p\ (x+2)) : \tau_4 \\
& \nwarrow_{\widehat{x}=\tau_3}\ (p\ (x+2)) : \tau_4 \\
& \nwarrow\ p : \tau_5 \to \tau_4,\ (x+2) : \tau_5 \\
& \nwarrow_{\widehat{p}=\tau_5\to\tau_4}\ (x+2) : \tau_5 \\
& \nwarrow_{\tau_5=int}\ x : int \\
& \nwarrow_{\widehat{x}=int}\ \square
\end{array}$$

So we have

$$\widehat{x} = \tau_1 = \tau_3 = \tau_5 = int \quad \tau_2 = (\tau_3 * \tau_4) = (int * \tau_4) \quad \tau = (\tau_1 \to \tau_2) = (int \to (int * \tau_4))$$

From which

$$\widehat{p} = \tau = (int \to (int * \tau_4)) \qquad \text{and} \qquad \widehat{p} = (\tau_5 \to \tau_4) = (int \to \tau_4)$$

Thus it must be the case that

$$int * \tau_4 = \tau_4$$

which is absurd, because it is not possible to unify τ_4 with a composite term containing an occurrence of τ_4. The above argument is represented more concisely below:

$$t = \textbf{rec}\ p.\quad \lambda\, \underset{int}{\square}\,.\ (\underset{int}{\square}, (\ \underset{int\to\tau_4}{\square}\ \ (\underset{int}{\square} + \underset{int}{\square})))$$

$$\underbrace{\underbrace{\underbrace{\underbrace{(int\to(int*\tau_4)) = (int\to\tau_4)}_{int*\tau_4} \Rightarrow \tau_4 = (int*\tau_4)}_{\tau_4}}_{int}}_{int}$$

So we have no solutions, and the term is not a well-formed term.

7.2 Operational Semantics of HOFL

In Section 6.1 we have defined the concepts of free variables and substitution for the λ-calculus. Now we define the same concepts in the case of HOFL, which will be necessary to define its operational semantics.

Definition 7.4 (Free variables). We define the set of free variables of HOFL terms by structural recursion, as follows:

$$\begin{aligned}
\mathrm{fv}(n) &\stackrel{\mathrm{def}}{=} \varnothing \\
\mathrm{fv}(x) &\stackrel{\mathrm{def}}{=} \{x\} \\
\mathrm{fv}(t_0\ \mathrm{op}\ t_1) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t) \cup \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}((t_0, t_1)) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{fst}(t)) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t) \\
\mathrm{fv}(\textbf{snd}(t)) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t) \\
\mathrm{fv}(\lambda x.\ t) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t) \backslash \{x\} \\
\mathrm{fv}((t_0\ t_1)) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{rec}\ x.\ t) &\stackrel{\mathrm{def}}{=} \mathrm{fv}(t) \backslash \{x\}
\end{aligned}$$

Finally as we did for λ-calculus we define the substitution operator in HOFL.

Definition 7.5 (Capture-avoiding substitution). Capture avoiding substitution $[{}^{t}/_{x}]$ of x with t is defined by structural recursion over HOFL terms as follows:

$$
\begin{aligned}
n[^t/_x] &= n \\
y[^t/_x] &\stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(t_0 \text{ op } t_1)[^t/_x] &\stackrel{\text{def}}{=} t_0[^t/_x] \text{ op } t_1[^t/_x] \quad \text{with op} \in \{+,-,\times\} \\
(\textbf{if } t' \textbf{ then } t_0 \textbf{ else } t_1)[^t/_x] &\stackrel{\text{def}}{=} \textbf{if } t'[^t/_x] \textbf{ then } t_0[^t/_x] \textbf{ else } t_1[^t/_x] \\
(t_0,t_1)[^t/_x] &\stackrel{\text{def}}{=} (t_0[^t/_x], t_1[^t/_x]) \\
\textbf{fst}(t')[^t/_x] &\stackrel{\text{def}}{=} \textbf{fst}(t'[^t/_x]) \\
\textbf{snd}(t')[^t/_x] &\stackrel{\text{def}}{=} \textbf{snd}(t'[^t/_x]) \\
(t_0\ t_1)[^t/_x] &\stackrel{\text{def}}{=} (t_0[^t/_x]\ t_1[^t/_x]) \\
(\lambda y.\ t')[^t/_x] &\stackrel{\text{def}}{=} \lambda z.\ (\ t'[^z/_y][^t/_x]\ ) \quad \text{for } z \notin \text{fv}(\lambda y.\ t') \cup \text{fv}(t) \cup \{x\} \\
(\textbf{rec } y.\ t')[^t/_x] &\stackrel{\text{def}}{=} \textbf{rec } z.\ (t'[^z/_y][^t/_x]) \quad \text{for } z \notin \text{fv}(\textbf{rec } y.\ t') \cup \text{fv}(t) \cup \{x\}
\end{aligned}
$$

Note that in the last two rules we perform α-conversion $[^z/_y]$ of the bound variable y with a fresh identifier z before the substitution. This ensures that the free occurrences of y in t, if any, are not bound accidentally after the substitution. As discussed in Section 6.1, the substitution is well defined if we consider the terms up to α-conversion (i.e., up to the renaming of bound variables). Obviously, we would like to extend these concepts to typed terms. So we are interested in understanding how substitution and α-conversion interact with typing. We have the following results.

Theorem 7.1 (Substitution respects types). *Let $x,t : \tau$ and $t' : \tau'$. Then, we have*

$$t'[^t/_x] : \tau'$$

Proof. The proof is in two steps. First we prove by rule induction the stronger predicate (for any term t' and type τ')

$$P(t' : \tau') \stackrel{\text{def}}{=} \forall x,t : \tau.\ \forall n \in \mathbb{N}.\ \forall x_1,z_1 : \tau_1,...,x_n,z_n : \tau_n.\ t'[^{z_1}/_{x_1},\cdots,^{z_n}/_{x_n},^t/_x] : \tau'$$

Second, the main statement of the theorem follows as the special case where $n = 0$. The stronger assertion is needed for handling the cases of functions (i.e., $t' = \lambda y.\ t''$ for some y and t'') and recursive expressions (i.e., $t' = \textbf{rec } y.\ t''$ for some y and t''), which are the only non-trivial cases (because of the way in which capture-avoiding substitution is defined). The reader is left to fill in the missing details of the proof as an exercise. □

We are now ready to present the operational semantics of HOFL. Unlike IMP, the operational semantics of HOFL is a simple manipulation of terms. This means that the operational semantics of HOFL defines a method to calculate the *canonical form* of a given term of HOFL. In particular, we focus on *closed terms* only, i.e., terms t with no free variables ($\text{fv}(t) = \varnothing$). Canonical forms are particular closed terms,

which we will assume to be the results of calculations (i.e., as ordinary values). For each type we fix the set of terms in canonical form by taking a subset of terms which reasonably represent the notion of values for that type.

As shown in the previous section, HOFL has three type constructors: the constant *int*, and the binary operators $*$ for pairs and $\rightarrow$ for functions. Terms which represent the integers provide the obvious canonical forms for the integer type. For pair types we take any pair of terms as canonical form: note that this choice is arbitrary; for example we could have taken instead pairs of terms that are themselves in canonical form. We will explain later the rationale of our choice. Finally, since HOFL is a higher-order language, functions are values. So it is quite natural to take all abstractions as canonical forms for the arrow type.

Definition 7.6 (Canonical forms). Let us define a set C_τ of canonical forms for each type τ as follows:

$$\frac{}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.\, t : \tau_0 \rightarrow \tau_1 \quad \lambda x.\, t \text{ closed}}{\lambda x.\, t \in C_{\tau_0 \rightarrow \tau_1}}$$

We now define the rules of the operational semantics; these rules define an evaluation relation:

$$t \rightarrow c$$

where t is a well-formed closed term of HOFL and c is its canonical form.

For terms that are already in canonical form according to Definition 7.6 we let

$$\frac{}{c \rightarrow c}$$

For clarity, the above rule offers a concise representation of the otherwise verbose rules

$$\frac{}{n \rightarrow n} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \rightarrow (t_0, t_1)} \qquad \frac{\lambda x.\, t : \tau_0 \rightarrow \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.\, t \rightarrow \lambda x.\, t}$$

Next, we give the rules for arithmetic expressions:

$$\frac{t_0 \rightarrow n_0 \quad t_1 \rightarrow n_1}{t_0 \text{ op } t_1 \rightarrow n_0 \underline{\text{op}}\, n_1} \qquad \frac{t \rightarrow 0 \quad t_0 \rightarrow c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \rightarrow c_0} \qquad \frac{t \rightarrow n \quad n \neq 0 \quad t_1 \rightarrow c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \rightarrow c_1}$$

For the arithmetic operators the semantics is obviously the simple application of the corresponding meta-operator as in IMP. Only, here we distinguish between HOFL syntactic operators and meta-operators by underlying the latter. For instance, we have $1 + 2 \rightarrow 3$, since $1 \rightarrow 1$, $2 \rightarrow 2$ and $1 \underline{+} 2 = 3$.

We recall that for the conditional statement, since we have no boolean values, we use the convention that **if** t **then** t_0 **else** t_1 stands for **if** $t = 0$ **then** t_0 **else** t_1, so the premise $t \rightarrow n \neq 0$ means the test is false and $t \rightarrow 0$ means the test is true.

Let us now consider pairing. Obviously, since we consider pairs as canonical values, we do not have to add further rules for simple pairs. We have instead two rules for projections:

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\mathbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\mathbf{snd}(t) \to c_1}$$

The rules are obviously similar: the canonical form of t is computed, which must be of the form (t_0, t_1), because t must have pair type for the projection to be applicable and $\mathbf{fst}(t)$ (and $\mathbf{snd}(t)$) typable. Note however that t_0 and t_1 need not be in canonical form. So only the canonical form of the component indicated by the projection operator is computed, with the other component discarded.

Function abstraction is handled by the axiom for terms already in canonical form, as in the case of pairing. For function application, we show two rules, according to two different evaluation strategies, called *lazy* and *eager*. In the lazy operational semantics, we do not evaluate the canonical forms of the parameters when passing them to the function body. The lazy semantics will be our primary focus in the rest of this part of the book concerned with HOFL:

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1\ t_0) \to c} \quad \text{(lazy)}$$

We remark that in the second premise of the rule, we replace each occurrence of x in t_1' with t_0, i.e., we replace each instance of x with a copy of the (non-evaluated) parameter t_0 and not with its canonical form.

For the sake of discussion let us consider the *eager* alternative to this rule:

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_0 \to c_0 \quad t_1'[{}^{c_0}/_x] \to c}{(t_1\ t_0) \to c} \quad \text{(eager)}$$

Unlike the lazy semantics, the eager semantics evaluates the parameters only once and does so before the substitution. Note that these two types of evaluation are not equivalent. If the evaluation of the argument does not terminate, and it is not needed, the lazy rule will guarantee convergence, while the eager rule will diverge. Vice versa, according to the lazy semantics, if the argument is actually needed it may later be evaluated several times (every time it is used).

Finally, we have a last rule for recursive terms:

$$\frac{t[{}^{\mathbf{rec}\ x.\ t}/_x] \to c}{\mathbf{rec}\ x.\ t \to c}$$

To evaluate the canonical form of **rec** $x.\ t$ we first plug into t the recursive definition itself in place of every occurrence of x and then compute the canonical form.

Example 7.6. Let us consider the term $t \stackrel{\text{def}}{=} \lambda x.\ 0+x$. Clearly the term t is closed and typable, with $t : \mathit{int} \to \mathit{int}$. It is already in canonical form and we have in fact

$$t \to c \ \nwarrow_{c=\lambda x.\ 0+x} \Box$$

Example 7.7. Let us consider the term $t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ 0+x$. Clearly the term t is closed and typable, with $t : \mathit{int}$. We show that the term has no canonical form, in fact

$$\begin{array}{rl} t \to c & \nwarrow\ (0+x)[{}^{t}/_{x}] \to c \\ & =\ 0+t \to c \\ \nwarrow_{c=c_1+c_2} & 0 \to c_1,\ t \to c_2 \\ \nwarrow_{c_1=0} & t \to c_2 \\ \nwarrow & \ldots \end{array}$$

Let us see an example which illustrates how rules are used to evaluate a function application.

Example 7.8 (Factorial). Let us consider the well-formed factorial function seen in Example 7.1:

$$\mathit{fact} \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f(x-1))$$

It is immediate to see that *fact* is closed and we know it has type $\mathit{int} \to \mathit{int}$. So we can calculate its canonical form by using the last rule seen and the axiom for terms in canonical form:

$$\frac{\lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathit{fact}(x-1)) \to \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathit{fact}(x-1))}{\mathit{fact} \to \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathit{fact}(x-1))}$$

We can apply this function to a specific value and calculate the canonical form of the result. For example, we see what is the canonical form c of the (closed and typable) term $(\mathit{fact}\ 2) : \mathit{int}$

$$
\begin{array}{rl}
(\mathit{fact}\ 2) \to c & \nwarrow\ \mathit{fact} \to \lambda x'.\, t',\quad t'[^{2}/_{x'}] \to c \\
& \nwarrow\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\mathit{fact}(x-1)) \to \lambda x'.\, t', \\
& \quad t'[^{2}/_{x'}] \to c \\
\nwarrow_{x'=x,\ t'=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times \mathit{fact}(x-1)} & \textbf{if}\ 2\ \textbf{then}\ 1\ \textbf{else}\ 2 \times (\mathit{fact}(2-1)) \to c \\
\nwarrow^{*} & 2 \times (\mathit{fact}(2-1)) \to c \\
\nwarrow_{c=c_1 \underline{\times} c_2} & 2 \to c_1,\quad (\mathit{fact}(2-1)) \to c_2 \\
\nwarrow^{*}_{c_1=2} & \mathit{fact} \to \lambda x''.\, t'',\quad \underbrace{t''[^{2-1}/_{x''}] \to c_2}_{\text{note that } 2-1 \text{ is not evaluated}} \\
\nwarrow_{x''=x,\ t''=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times \mathit{fact}(x-1)} & \textbf{if}\ (2-1)\ \textbf{then}\ 1 \\
& \textbf{else}\ (2-1) \times (\mathit{fact}((2-1)-1)) \to c_2 \\
& \nwarrow\ 2-1 \to n,\quad n \neq 0, \\
& \quad (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow_{n=n_1 \underline{-} n_2} & 2 \to n_1,\quad 1 \to n_2,\quad n_1 \underline{-} n_2 \neq 0, \\
& \quad (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow^{*}_{n_1=2,\ n_2=1} & (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow^{*}_{c_2=c_3 \underline{\times} c_4} & 2-1 \to c_3,\quad \mathit{fact}((2-1)-1) \to c_4 \\
\nwarrow^{*}_{c_3=1} & \mathit{fact}((2-1)-1) \to c_4 \\
\nwarrow^{*} & \textbf{if}\ (2-1)-1\ \textbf{then}\ 1\ \textbf{else} \\
& \quad ((2-1)-1) \times (\mathit{fact}(((2-1)-1)-1)) \to c_4 \\
& \nwarrow\ (2-1)-1 \to 0,\quad 1 \to c_4 \\
\nwarrow^{*}_{c_4=1} & \square
\end{array}
$$

So we have

$$c = c_1 \underline{\times} c_2 = 2 \underline{\times} (c_3 \underline{\times} c_4) = 2 \underline{\times} (1 \underline{\times} 1) = 2$$

Example 7.9 (Lazy vs eager evaluation). The aim of this example is to illustrate the difference between lazy and eager semantics. Let us consider the term

$$t \stackrel{\text{def}}{=} ((\lambda x : \mathit{int}.\ 3)(\textbf{rec}\ y : \mathit{int}.\ y)) : \mathit{int}$$

also written more concisely as

$$t \stackrel{\text{def}}{=} (\lambda x.\ 3)\, \textbf{rec}\ y.\ y$$

assuming $\widehat{x} = \widehat{y} = \mathit{int}$. It consists of the constant function $\lambda x.\ 3$ applied to a diverging term **rec** $y.\ y$ (i.e., a term with no canonical form).

- **Lazy evaluation**
  Lazy evaluation evaluates a parameter only if needed: if a parameter is never used in a function or in a specific instance of a function it will never be evaluated. Let us show our example:

$$\begin{array}{lll} ((\lambda x.\,3)\,\mathbf{rec}\ y.\,y) \to c & \nwarrow & \lambda x.\,3 \to \lambda x.\,t, \quad t[^{\mathbf{rec}\ y.\,y}/_x] \to c \\ & \nwarrow_{t=3} & 3[^{\mathbf{rec}\ y.\,y}/_x] \to c \\ & \nwarrow_{c=3} & \Box \end{array}$$

So although the argument **rec** *y. y* has no canonical form the application can be evaluated.

- **Eager evaluation**
  On the contrary in the eager semantics this term has no canonical form since the parameter must be evaluated before the application, leading to a diverging computation:

$$\begin{array}{lll} ((\lambda x.\,3)\,\mathbf{rec}\ y.\,y) \to c & \nwarrow & \lambda x.\,3 \to \lambda x.\,t, \quad \mathbf{rec}\ y.\,y \to c_1, \quad t[^{c_1}/_x] \to c \\ & \nwarrow_{t=3} & \mathbf{rec}\ y.\,y \to c_1, \quad 3[^{c_1}/_x] \to c \\ & \nwarrow & \mathbf{rec}\ y.\,y \to c_1, \quad 3[^{c_1}/_x] \to c \\ & \nwarrow & \ldots \end{array}$$

  So the evaluation does not terminate.

However if the parameter of a function is used n times, the parameter will be evaluated n times (at most) in the lazy semantics and only once in the eager case.

We conclude this chapter by presenting a theorem that guarantees that

1. if a term can be reduced to a canonical form then it is unique (determinacy);
2. the evaluation of the canonical form preserves the type assignments (type preservation).

Theorem 7.2. *Let t be a closed and typable term.*

1. *For any canonical forms c, c', if $t \to c$ and $t \to c'$ then $c = c'$.*
2. *For any canonical form c and type τ, if $t \to c$ and $t : \tau$ then $c : \tau$.*

Proof. Property 1 is proved by rule induction, taking the predicate

$$P(t \to c) \stackrel{\text{def}}{=} \forall c'.\, t \to c' \Rightarrow c = c'$$

We show only the case of the application rule; the remainder of the proof of the theorem, including the proof of Property 2, is left as an exercise (see Problem 7.11). We have the rule

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

We assume the inductive hypotheses

- $P(t_1 \to \lambda x.\, t_1') \stackrel{\text{def}}{=} \forall c'.\, t_1 \to c' \Rightarrow \lambda x.\, t_1' = c'$
- $P(t_1'[^{t_0}/_x] \to c) \stackrel{\text{def}}{=} \forall c'.\, t_1'[^{t_0}/_x] \to c' \Rightarrow c = c'$

We want to prove

$$P((t_1\ t_0) \to c) \stackrel{\text{def}}{=} \forall c'.\ (t_1\ t_0) \to c' \Rightarrow c = c'$$

As usual, we assume the premise of the implication:

$$(t_1\ t_0) \to c'$$

From it, by goal reduction

$$(t_1\ t_0) \to c' \quad \nwarrow \quad t_1 \to \lambda x'.\ t_1'', \quad t_1''[{}^{t_0}/_{x'}] \to c'$$

Then we have by the first inductive hypothesis

$$\lambda x.\ t_1' = \lambda x'.\ t_1''$$

i.e., $x = x'$ and $t_1' = t_1''$. Then $t_1''[{}^{t_0}/_{x'}] = t_1'[{}^{t_0}/_{x}]$ and by the second inductive hypothesis we have $c = c'$. □

Problems

7.1. Let $x, y, w : \mathit{int}$, and $f : \mathit{int} \to (\mathit{int} \to \mathit{int})$. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ (\lambda y.\ (y + w))\ \textbf{else}\ (f\ w)$$

1. Compute the term $t[{}^{((f\ x)\ y)}/_{w}]$.
2. Compute the term $t[{}^{((f\ x)\ y)}/_{x}]$.

Hint: The exercise is about practice with capture-avoiding substitutions. You are allowed to introduce additional (typed) variables if needed.

7.2. Is it possible to assign a type to the HOFL pre-term below? If yes, compute its principal type.

$$\textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{snd}(x)\ \textbf{then}\ 1\ \textbf{else}\ f(\ \textbf{fst}(x)\ ,\ (\textbf{fst}(x)\ \ \textbf{snd}(x)))$$

7.3. A *list of positive numbers* is defined by the following syntax, where $n \in \mathbb{N}, n > 0$:

$$L\ ::=\ (n, 0)\ \mid\ (n, L)$$

For instance the list with 3 followed by 5 is represented by the term $(3, (5, 0))$.

1. Define a HOFL term t (closed and typable) such that the application $(t\,L)$ to a list L of three elements returns the last element of the list.
2. Is it possible to find a closed and typable HOFL term which returns the last element of a generic list?

7.4. Given the two HOFL terms

$$t_1 \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, x+3 \qquad t_2 \stackrel{\text{def}}{=} \lambda z.\, \mathbf{fst}(z)+3$$

1. Compute their types.
2. Prove that, given the canonical form $c : \tau$, the two terms

$$((t_1\ 1)\ c) \qquad \text{and} \qquad (t_2\ (1,c))$$

yield the same canonical form.

7.5. Consider the HOFL term

$$map \stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, ((f\ \mathbf{fst}(x)),(f\ \mathbf{snd}(x)))$$

Show that *map* is a typable term and give its principal type. Then, compute the canonical form of the term

$$((map\ (\lambda x.\, 2\times x))\ (1,2))$$

7.6. Determine the type of the HOFL term

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\, ((\lambda y.\ \mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0)\ x).$$

Then compute its operational semantics.

7.7. Recall the definition of *binomial coefficients* $\binom{n}{k}$ from Problem 4.13:

$$\binom{n}{0} \stackrel{\text{def}}{=} 1 \qquad \binom{n}{n} \stackrel{\text{def}}{=} 1 \qquad \binom{n+1}{k+1} \stackrel{\text{def}}{=} \binom{n}{k} + \binom{n}{k+1}.$$

where $n,k \in \mathbb{N}$ and $0 \le k \le n$. Consider the corresponding HOFL program:

$$\begin{aligned} t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\, \lambda n.\, \lambda k.\ & \mathbf{if}\ k\ \mathbf{then}\ 1 \\ & \mathbf{else\ if}\ n-k\ \mathbf{then}\ 1 \\ & \quad \mathbf{else}\ ((f\ (n-1))\ k)+((f\ (n-1))\ (k-1)). \end{aligned}$$

Compute its type and evaluate the canonical form of the term $((t\ 2)\ 1)$.

7.8. Consider the Fibonacci sequence already found in Problem 4.14

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1)+F(n)$$

where $n \in \mathbb{N}$.

1. Write a well-formed, closed HOFL term $t : int \to int$ to compute F.
2. Compute the operational semantics of $(t\ 2)$

7.9. Check whether the HOFL pre-term

$$\lambda x.\ \lambda y.\ \lambda z.\ \textbf{if}\ z\ \textbf{then}\ (y\ x)\ \textbf{else}\ (x\ y).$$

is typable, in which case give its type.

7.10. Consider the HOFL pre-term $t = \lambda x.\ (x\ x)$. Prove that it is not typable. Try to compute anyway the canonical form of the application $(t\ t)$. Given that any well-typed term without recursion has a canonical form, argue why the given term is not typable.

7.11. Complete the proof of Theorem 7.2.

7.12. Suppose we extend HOFL with the inference rule

$$\frac{t_1 \to 0}{t_1 \times t_2 \to 0}$$

Prove that the property of determinacy

$$\forall t, c_1, c_2.\ t \to c_1\ \wedge\ t \to c_2 \Rightarrow c_1 = c_2$$

is still valid. What if also the inference rule below is added?

$$\frac{t_2 \to 0}{t_1 \times t_2 \to 0}$$

7.13. Prove that typable terms are uniquely typed, i.e., that for any pre-term t and types τ, τ', if $t : \tau$ and $t : \tau'$ then $\tau = \tau'$.

Chapter 8
Domain Theory

Order, unity and continuity are human inventions just as truly as catalogues and encyclopedias. (Bertrand Russell)

Abstract As we have done for IMP, we would like to introduce the denotational semantics of HOFL, for which we need to develop a proper domain theory that is more sophisticated than the one presented in Chapter 5. In order to define the denotational semantics of IMP, we have shown that the semantic domain of commands, for which we need to apply the fixpoint theorem, has the required properties. The situation is more complicated for HOFL, because HOFL provides constructors for infinitely many term types, so there are infinitely many domains to be considered. We will handle this problem by showing, using structural induction, that the type constructors of HOFL correspond to domains which are equipped with adequate $CPO_\perp$ structures and that we can define useful continuous functions between them.

8.1 The Flat Domain of Integer Numbers $\mathbb{Z}_\perp$

The first domain we introduce is very simple: it consists of all the integer numbers together with a distinguished bottom element. It relies on a flat order in the sense of Example 5.5.

Definition 8.1 ($\mathbb{Z}_\perp$). We define the CPO with bottom $\mathbb{Z}_\perp = (\mathbb{Z} \cup \{\perp\}, \sqsubseteq)$ as follows:

- $\mathbb{Z}$ is the set of integer numbers;
- $\perp$ is a distinguished bottom element;
- $\forall x \in \mathbb{Z} \cup \{\perp\}.\ \perp \sqsubseteq x$ and $x \sqsubseteq x$.

It is immediate to check that $\mathbb{Z}_\perp$ is a CPO with bottom, where $\perp$ is the bottom element and each chain has a lub because chains are all finite: they contain either one or two different elements.

Remark 8.1. Since in this chapter we present several different domains, each coming with its proper order relation and bottom element, we find it useful to annotate them with the name of the domain as a subscript to avoid ambiguities. For example, we can write $\perp_{\mathbb{Z}_\perp}$ to make explicit that we are referring to the bottom element of the

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_8

domain $\mathbb{Z}_\perp$. Also note that the subscript $\perp$ we attach to the name of the domain $\mathbb{Z}$ is just a tag and it should not be confused with the name of the bottom element itself: it is the standard way to indicate that the domain $\mathbb{Z}$ is enriched with a bottom element (e.g., we could have used a different notation such as $\underline{\mathbb{Z}}$ to the same purpose).

8.2 Cartesian Product of Two Domains

Given two CPO$_\perp$s we can combine them to obtain another CPO$_\perp$ whose elements are pairs formed with one element from each CPO$_\perp$.

Definition 8.2 (Cartesian product domain). Let

$$\mathscr{D} = (D, \sqsubseteq_D) \qquad \mathscr{E} = (E, \sqsubseteq_E)$$

be two CPO$_\perp$s. Now we define their cartesian product domain

$$\mathscr{D} \times \mathscr{E} = (D \times E, \sqsubseteq_{D\times E})$$

1. whose elements are pairs of elements from D and E; and
2. whose order $\sqsubseteq_{D\times E}$ is defined as follows:[1]

$$\forall d_0, d_1 \in D,\ \forall e_0, e_1 \in E.\ (d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1) \Leftrightarrow d_0 \sqsubseteq_D d_1 \wedge e_0 \sqsubseteq_E e_1$$

Proposition 8.1. *$(D \times E, \sqsubseteq_{D\times E})$ is a partial order with bottom.*

Proof. We need to show that the relation $\sqsubseteq_{D\times E}$ is reflexive, antisymmetric and transitive:

reflexivity: since $\sqsubseteq_D$ and $\sqsubseteq_E$ are reflexive we have $\forall e \in E.\ e \sqsubseteq_E e$ and $\forall d \in D.\ d \sqsubseteq_D d$ so by definition of $\sqsubseteq_{D\times E}$ we have

$$\forall d \in D\ \forall e \in E.\ (d, e) \sqsubseteq_{D\times E} (d, e).$$

antisymmetry: let us assume $(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D\times E} (d_0, e_0)$ so by definition of $\sqsubseteq_{D\times E}$ we have $d_0 \sqsubseteq_D d_1$ (using the first relation) and $d_1 \sqsubseteq_D d_0$ (by using the second relation) so it must be that $d_0 = d_1$ and similarly $e_0 = e_1$, hence $(d_0, e_0) = (d_1, e_1)$.

transitivity: let us assume $(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D\times E} (d_2, e_2)$. By definition of $\sqsubseteq_{D\times E}$ we have $d_0 \sqsubseteq_D d_1$, $d_1 \sqsubseteq_D d_2$, $e_0 \sqsubseteq_E e_1$ and $e_1 \sqsubseteq_E e_2$. By transitivity of $\sqsubseteq_D$ and $\sqsubseteq_E$ we have $d_0 \sqsubseteq_D d_2$ and $e_0 \sqsubseteq_E e_2$. By definition of $\sqsubseteq_{D\times E}$ we get $(d_0, e_0) \sqsubseteq_{D\times E} (d_2, e_2)$.

Finally, we show that there is a bottom element. Let $\perp_{D\times E} = (\perp_D, \perp_E)$. In fact $\forall d \in D, e \in E.\ \perp_D \sqsubseteq d \wedge \perp_E \sqsubseteq e$, thus $(\perp_D, \perp_E) \sqsubseteq_{D\times E} (d, e)$. □

[1] Note that the order is different from the lexicographic one considered in Example 4.9.

It remains to show the completeness of $\mathscr{D} \times \mathscr{E}$.

Theorem 8.1 (Completeness of $\mathscr{D} \times \mathscr{E}$). *The PO $\mathscr{D} \times \mathscr{E}$ defined above is complete.*

Proof. We prove that for each chain $\{(d_i, e_i)\}_{i \in \mathbb{N}}$ it holds that

$$\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) = \left(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i \right)$$

Obviously $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$ is an upper bound, indeed for each $j \in \mathbb{N}$ we have $d_j \sqsubseteq_D \bigsqcup_{i \in \mathbb{N}} d_i$ and $e_j \sqsubseteq_E \bigsqcup_{i \in \mathbb{N}} e_i$ so by definition of $\sqsubseteq_{D \times E}$ it holds that $(d_j, e_j) \sqsubseteq_{D \times E} (\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$.
Moreover $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$ is also the least upper bound. Indeed, let (d, e) be an upper bound of $\{(d_i, e_i)\}_{i \in \mathbb{N}}$; since $\bigsqcup_{i \in \mathbb{N}} d_i$ is the lub of $\{d_i\}_{i \in \mathbb{N}}$ we have $\bigsqcup_{i \in \mathbb{N}} d_i \sqsubseteq_D d$, furthermore we have that $\bigsqcup_{i \in \mathbb{N}} e_i$ is the lub of $\{e_i\}_{i \in \mathbb{N}}$ so $\bigsqcup_{i \in \mathbb{N}} e_i \sqsubseteq_E e$. So by definition of $\sqsubseteq_{D \times E}$ we have $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i) \sqsubseteq_{D \times E} (d, e)$. Thus $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$ is the least upper bound. □

We can now define suitable projection operators over $\mathscr{D} \times \mathscr{E}$.

Definition 8.3 (Projection operators π_1 and π_2). Let $(d, e) \in D \times E$ be a pair. We define the left and right projection functions $\pi_1 : D \times E \to D$ and $\pi_2 : D \times E \to E$ as follows:

$$\pi_1((d, e)) \stackrel{\text{def}}{=} d \qquad \text{and} \qquad \pi_2((d, e)) \stackrel{\text{def}}{=} e.$$

Recall that in order to use a function in domain theory we have to show that it is continuous; this ensures that the function respects the domain structure (i.e., the function preserves the order and limits) and so we can calculate its fixpoints to solve recursive equations. So we have to prove that each function which we use on $\mathscr{D} \times \mathscr{E}$ is continuous. The proof that projections are monotone is immediate and left as an exercise (see Problem 8.1).

Theorem 8.2 (Continuity of π_1 and π_2). *Let π_1 and π_2 be the projection functions in Definition 8.3 and let $\{(d_i, e_i)\}_{i \in \mathbb{N}}$ be a chain of elements in $\mathscr{D} \times \mathscr{E}$, then*

$$\pi_1 \left(\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) \right) = \bigsqcup_{i \in \mathbb{N}} \pi_1((d_i, e_i)) \qquad \pi_2 \left(\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) \right) = \bigsqcup_{i \in \mathbb{N}} \pi_2((d_i, e_i))$$

Proof. Let us prove the first statement:

$$\begin{aligned} \pi_1 \left(\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) \right) &= \pi_1 \left(\left(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i \right) \right) && \text{(by definition of limit in } D \times E) \\ &= \bigsqcup_{i \in \mathbb{N}} d_i && \text{(by definition of projection)} \\ &= \bigsqcup_{i \in \mathbb{N}} \pi_1((d_i, e_i)) && \text{(by definition of projection)} \end{aligned}$$

For the second statement the proof is completely analogous. □

8.3 Functional Domains

Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be two CPOs. In the following we denote by $D \to E \stackrel{\text{def}}{=} \{f \mid f : D \to E\}$ the set of all functions from D to E (where the order relations are not important), while we denote by $[D \to E] \subseteq D \to E$ the set of all continuous functions from D to E (i.e., $[D \to E]$ contains just the functions that preserve order and limits). As for cartesian product, we can define a suitable order on the set $[D \to E]$ to get a $\text{CPO}_\perp$. Note that as usual we require the continuity of the functions to preserve the applicability of fixpoint theory.

Definition 8.4 (Continuous functions domain). Let us consider the $\text{CPO}_\perp$s

$$\mathscr{D} = (D, \sqsubseteq_D) \qquad \mathscr{E} = (E, \sqsubseteq_E)$$

We define an order on the set of continuous functions from D to E as follows:

$$[\mathscr{D} \to \mathscr{E}] = \left([D \to E], \sqsubseteq_{[D \to E]}\right)$$

where

1. $[D \to E] = \{f \mid f : D \to E, f \text{ is continuous}\}$
2. $f \sqsubseteq_{[D \to E]} g \Leftrightarrow \forall d \in D. f(d) \sqsubseteq_E g(d)$

We leave as an exercise the proof that $[\mathscr{D} \to \mathscr{E}]$ is a PO with bottom, namely that the relation $\sqsubseteq_{[D \to E]}$ is reflexive, antisymmetric and transitive and that the function $\perp_{[D \to E]} : D \to E$ defined by letting, for any $d \in D$

$$\perp_{[D \to E]}(d) \stackrel{\text{def}}{=} \perp_E$$

is continuous and that it is also the bottom element of $[\mathscr{D} \to \mathscr{E}]$ (see Problem 8.2).

We show that the PO $[\mathscr{D} \to \mathscr{E}]$ is complete. In order to simplify the proof we first introduce the following lemmas.

Lemma 8.1 (Switch lemma). *Let $(E, \sqsubseteq_E)$ be a CPO whose elements are of the form $e_{n,m}$ with $n, m \in \mathbb{N}$. If $\sqsubseteq_E$ is such that*

$$e_{n,m} \sqsubseteq_E e_{n',m'} \text{ if } n \leq n' \text{ and } m \leq m'$$

then, it holds

$$\bigsqcup_{n,m \in \mathbb{N}} e_{n,m} = \bigsqcup_{n \in \mathbb{N}} \left(\bigsqcup_{m \in \mathbb{N}} e_{n,m} \right) = \bigsqcup_{m \in \mathbb{N}} \left(\bigsqcup_{n \in \mathbb{N}} e_{n,m} \right) = \bigsqcup_{k \in \mathbb{N}} e_{k,k}$$

Proof. The relation between the elements of E can be summarised as follows:

$$
\begin{array}{ccccccccc}
\vdots & & \vdots & & \vdots & & \vdots & & \iddots \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \\
e_{n,0} & \sqsubseteq & e_{n,1} & \sqsubseteq & e_{n,2} & \sqsubseteq \cdots \sqsubseteq & e_{n,m} & \sqsubseteq & \cdots \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \\
\vdots & & \vdots & & \vdots & \iddots & \vdots & & \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \\
e_{2,0} & \sqsubseteq & e_{2,1} & \sqsubseteq & e_{2,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq & \cdots \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \\
e_{1,0} & \sqsubseteq & e_{1,1} & \sqsubseteq & e_{1,2} & \sqsubseteq \cdots \sqsubseteq & e_{1,m} & \sqsubseteq & \cdots \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \\
e_{0,0} & \sqsubseteq & e_{0,1} & \sqsubseteq & e_{0,2} & \sqsubseteq \cdots \sqsubseteq & e_{0,m} & \sqsubseteq & \cdots
\end{array}
$$

We show that all the following sets have the same upper bounds:

$$
\{e_{n,m}\}_{n,m\in\mathbb{N}} \qquad \left\{\bigsqcup_{m\in\mathbb{N}} e_{n,m}\right\}_{n\in\mathbb{N}} \qquad \left\{\bigsqcup_{n\in\mathbb{N}} e_{n,m}\right\}_{m\in\mathbb{N}} \qquad \{e_{k,k}\}_{k\in\mathbb{N}}
$$

- Let us consider the first two sets. For any $n \in \mathbb{N}$, let $e_n = \bigsqcup_{j\in\mathbb{N}} e_{n,j}$. This amounts to considering each row of the above diagram and computing the least upper bound for the elements in the same row. Clearly, $e_{n_1} \sqsubseteq e_{n_2}$ when $n_1 \leq n_2$ because for any $j \in \mathbb{N}$ an upper bound of $e_{n_2,j}$ is also an upper bound of $e_{n_1,j}$.

$$
\begin{array}{ccccccccc}
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \sqcup| \\
e_{n,0} & \sqsubseteq & e_{n,1} & \sqsubseteq & e_{n,2} & \sqsubseteq \cdots \sqsubseteq & e_{n,m} & \sqsubseteq \cdots \sqsubseteq & e_n = \bigsqcup_{m\in\mathbb{N}} e_{n,m} \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \sqcup| \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \sqcup| \\
e_{2,0} & \sqsubseteq & e_{2,1} & \sqsubseteq & e_{2,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq \cdots \sqsubseteq & e_2 = \bigsqcup_{m\in\mathbb{N}} e_{2,m} \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \sqcup| \\
e_{1,0} & \sqsubseteq & e_{1,1} & \sqsubseteq & e_{1,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq \cdots \sqsubseteq & e_1 = \bigsqcup_{m\in\mathbb{N}} e_{1,m} \\
\sqcup| & & \sqcup| & & \sqcup| & & \sqcup| & & \sqcup| \\
e_{0,0} & \sqsubseteq & e_{0,1} & \sqsubseteq & e_{0,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq \cdots \sqsubseteq & e_0 = \bigsqcup_{m\in\mathbb{N}} e_{0,m}
\end{array}
$$

Let e be an upper bound of $\{e_i\}_{i\in\mathbb{N}}$. We want to show that e is an upper bound for $\{e_{n,m}\}_{n,m\in\mathbb{N}}$. Take any $n, m \in \mathbb{N}$. Then

$$
e_{n,m} \sqsubseteq \bigsqcup_{j\in\mathbb{N}} e_{n,j} = e_n \sqsubseteq e
$$

since $e_{n,m}$ is an element of the chain $\{e_{n,j}\}_{j\in\mathbb{N}}$ whose limit is $e_n = \bigsqcup_{j\in\mathbb{N}} e_{n,j}$. Thus e is an upper bound for $\{e_{n,m}\}_{n,m\in\mathbb{N}}$.

Vice versa, let e be an upper bound of $\{e_{i,j}\}_{i,j\in\mathbb{N}}$ and consider $e_n = \bigsqcup_{m\in\mathbb{N}} e_{n,m}$ for some n. Since $\{e_{n,m}\}_{m\in\mathbb{N}} \subseteq \{e_{i,j}\}_{i,j\in\mathbb{N}}$, obviously e is an upper bound for $\{e_{n,m}\}_{m\in\mathbb{N}}$ and therefore $e_n \sqsubseteq e$, because e_n is the lub of $\{e_{n,m}\}_{m\in\mathbb{N}}$.

- The correspondence between the sets of upper bounds of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ and $\{\bigsqcup_{n\in\mathbb{N}} e_{n,m}\}_{m\in\mathbb{N}}$ can be proved analogously.
- Finally, let us consider the sets $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ and $\{e_{k,k}\}_{k\in\mathbb{N}}$ and show that they have the same set of upper bounds.
  Take any $n,m \in \mathbb{N}$ and let $k = \max\{n,m\}$. We have

$$e_{n,m} \sqsubseteq e_{n,k} \sqsubseteq e_{k,k}$$

  thus any upper bound of $\{e_{k,k}\}_{k\in\mathbb{N}}$ is also an upper bound of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$.
  Vice versa, it is immediate to check that $\{e_{k,k}\}_{k\in\mathbb{N}}$ is a subset of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ so any upper bound of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ is also an upper bound of $\{e_{k,k}\}_{k\in\mathbb{N}}$.

We conclude by noting that the set of upper bounds $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ has a least element. In fact, $\{\bigsqcup_{m\in\mathbb{N}} e_{n,m}\}_{n\in\mathbb{N}}$ is a chain, and it has a lub because E is a CPO. □

Lemma 8.2. *Let $\{f_n\}_{n\in\mathbb{N}}$ be a chain of functions*[2] *in $\mathscr{D} \to \mathscr{E}$. Then the lub $\bigsqcup_{n\in\mathbb{N}} f_n$ exists and it is defined as*

$$\left(\bigsqcup_{n\in\mathbb{N}} f_n\right)(d) = \bigsqcup_{n\in\mathbb{N}} (f_n(d))$$

Proof. The function

$$h \stackrel{\text{def}}{=} \lambda d. \bigsqcup_{n\in\mathbb{N}} (f_n(d))$$

is clearly an upper bound for $\{f_n\}_{n\in\mathbb{N}}$ since for every $k \in \mathbb{N}$ and $d \in D$ we have $f_k(d) \sqsubseteq_E \bigsqcup_{n\in\mathbb{N}} f_n(d)$.

The function h is also the lub of $\{f_n\}_{n\in\mathbb{N}}$. In fact, given any other upper bound g, i.e., such that $f_n \sqsubseteq_{D\to E} g$ for any $n \in \mathbb{N}$, we have that for any $d \in D$ the element $g(d)$ is an upper bound of the chain $\{f_n(d)\}_{n\in\mathbb{N}}$ and therefore $\bigsqcup_{n\in\mathbb{N}}(f_n(d)) \sqsubseteq_E g(d)$. □

Lemma 8.3. *Let $\{f_n\}_{n\in\mathbb{N}}$ be a chain of continuous functions in $[\mathscr{D} \to \mathscr{E}]$ and let $\{d_n\}_{n\in\mathbb{N}}$ be a chain of $\mathscr{D}$. Then, the function*

$$h \stackrel{\text{def}}{=} \lambda d. \bigsqcup_{n\in\mathbb{N}} (f_n(d))$$

is continuous, namely

$$h\left(\bigsqcup_{m\in\mathbb{N}} d_m\right) = \bigsqcup_{m\in\mathbb{N}} h(d_m)$$

Furthermore, h is the lub of $\{f_n\}_{n\in\mathbb{N}}$ not only in $\mathscr{D} \to \mathscr{E}$ as stated by Lemma 8.2, but also in $[\mathscr{D} \to \mathscr{E}]$.

[2] Note that the f_n are not necessarily continuous, because we select $\mathscr{D} \to \mathscr{E}$ and not $[\mathscr{D} \to \mathscr{E}]$.

Proof.

$$
\begin{aligned}
h\left(\bigsqcup_{m\in\mathbb{N}} d_m\right) &= \bigsqcup_{n\in\mathbb{N}}\left(f_n\left(\bigsqcup_{m\in\mathbb{N}} d_m\right)\right) && \text{(by definition of } h\text{)}\\
&= \bigsqcup_{n\in\mathbb{N}}\left(\bigsqcup_{m\in\mathbb{N}} (f_n(d_m))\right) && \text{(by continuity of } f_n\text{)}\\
&= \bigsqcup_{m\in\mathbb{N}}\left(\bigsqcup_{n\in\mathbb{N}} (f_n(d_m))\right) && \text{(by Switch Lemma 8.1)}\\
&= \bigsqcup_{m\in\mathbb{N}} h(d_m) && \text{(by definition of } h\text{)}
\end{aligned}
$$

Note that, in the previous passages, the premises for applying the Switch Lemma 8.1 hold because the elements $\{f_n(d_m)\}_{n,m\in\mathbb{N}}$ are in the CPO E and they satisfy $f_n(d_m) \sqsubseteq_E f_{n'}(d_{m'})$ whenever $n \leq n'$ and $m \leq m'$ as $f_n(d_m) \sqsubseteq_E f_n(d_{m'})$ by monotonicity of f_n (because $d_m \sqsubseteq_D d_{m'}$) and $f_n(d_{m'}) \sqsubseteq_E f_{n'}(d_{m'})$ because $f_n \sqsubseteq_{[D\to E]} f_{n'}$. The upper bounds of $\{f_n\}_{n\in\mathbb{N}}$ in the PO $\mathscr{D} \to \mathscr{E}$ are a larger set than those in $[\mathscr{D} \to \mathscr{E}]$, thus if h is the lub in $\mathscr{D} \to \mathscr{E}$, it is also the lub in $[\mathscr{D} \to \mathscr{E}]$. □

Theorem 8.3 ($[\mathscr{D} \to \mathscr{E}]$ is a CPO$_\perp$). *The PO $[\mathscr{D} \to \mathscr{E}]$ is a CPO$_\perp$.*

Proof. The statement follows immediately from the previous lemmas. □

8.4 Lifting

In IMP we introduced a lifting operator (see Definition 6.9) on functions $f : \Sigma \to \Sigma_\perp$ to derive a function $f^* : \Sigma_\perp \to \Sigma_\perp$ defined over the lifted domain $\Sigma_\perp$, and thus able to handle the argument $\perp_{\Sigma_\perp}$. In the semantics of HOFL we need the same operator in a more general fashion: we need to apply the lifting operator to any domain, not just Σ.

Definition 8.5 (Lifted domain). Let $\mathscr{D} = (D, \sqsubseteq_D)$ be a CPO and let $\perp$ be an element not in D. We define the lifted domain $\mathscr{D}_\perp = (D_\perp, \sqsubseteq_{D_\perp})$ as follows:

- $D_\perp \stackrel{\text{def}}{=} \{\perp\} \uplus D = \{(0,\perp)\} \cup (\{1\} \times D)$
- $\perp_{D_\perp} \stackrel{\text{def}}{=} (0,\perp)$
- $\forall x \in D_\perp.\ \perp_{D_\perp} \sqsubseteq_{D_\perp} x$
- $\forall d_1, d_2 \in D.\ d_1 \sqsubseteq_D d_2 \Rightarrow (1,d_1) \sqsubseteq_{D_\perp} (1,d_2)$

We leave it as an exercise to show that $\mathscr{D}_\perp$ is a CPO$_\perp$ (see Problem 8.3).
We define a lifting function $\lfloor\cdot\rfloor : D \to D_\perp$ by letting, for any $d \in D$

$$\lfloor d \rfloor \stackrel{\text{def}}{=} (1,d)$$

As it was the case for Σ in the IMP semantics, when we add a bottom element to a domain $\mathscr{D}$ we would like to extend the continuous functions in $[D \to E]$ to continuous functions in $[D_\perp \to E]$. The function defining the extension should itself be continuous.

Definition 8.6 (Lifting). Let $\mathscr{D}$ be a CPO and let $\mathscr{E}$ be a $CPO_\perp$. We define the lifting operator $(\cdot)^* : [D \to E] \to [D_\perp \to E]$ as follows:

$$\forall f \in [D \to E]\,.\, f^*(x) \stackrel{\text{def}}{=} \begin{cases} \perp_E & \text{if } x = \perp_{D_\perp} \\ f(d) & \text{if } x = \lfloor d \rfloor \text{ for some } d \in D \end{cases}$$

We need to prove that the definition is well given and that the lifting operator is continuous.

Theorem 8.4. *Let $\mathscr{D}, \mathscr{E}$ be two CPOs.*

1. *If $f : D \to E$ is continuous, then f^* is continuous.*
2. *The operator $(\cdot)^*$ is continuous.*

Proof. We prove the two statements separately.

1. We need to prove that if $f \in [D \to E]$, then $f^* \in [D_\perp \to E]$. Let $\{x_n\}_{n\in\mathbb{N}}$ be a chain in $\mathscr{D}_\perp$. We have to prove $f^*(\bigsqcup_{n\in\mathbb{N}} x_n) = \bigsqcup_{n\in\mathbb{N}} f^*(x_n)$.
   If $\forall n \in \mathbb{N}.\ x_n = \perp_{D_\perp}$, then this is obvious.
   Otherwise, for some $k \in \mathbb{N}$ there must exist a set of elements $\{d_{n+k}\}_{n\in\mathbb{N}}$ in D such that for all $m \geq k$ we have $x_m = \lfloor d_m \rfloor$ and also $\bigsqcup_{n\in\mathbb{N}} x_n = \bigsqcup_{n\in\mathbb{N}} x_{n+k} = \lfloor \bigsqcup_{n\in\mathbb{N}} d_{n+k} \rfloor$ (by prefix independence of the limit, Lemma 5.1). Then

$$\begin{aligned}
f^*\left(\bigsqcup_{n\in\mathbb{N}} x_n\right) &= f^*\left(\left\lfloor \bigsqcup_{n\in\mathbb{N}} d_{n+k} \right\rfloor\right) && \text{by the above argument} \\
&= f\left(\bigsqcup_{n\in\mathbb{N}} d_{n+k}\right) && \text{by definition of lifting} \\
&= \bigsqcup_{n\in\mathbb{N}} f(d_{n+k}) && \text{by continuity of } f \\
&= \bigsqcup_{n\in\mathbb{N}} f^*(\lfloor d_{n+k} \rfloor) && \text{by definition of lifting} \\
&= \bigsqcup_{n\in\mathbb{N}} f^*(x_{n+k}) && \text{by definition of } x_{n+k} \\
&= \bigsqcup_{n\in\mathbb{N}} f^*(x_n) && \text{by Lemma 5.1}
\end{aligned}$$

2. We leave the proof that $(\cdot)^*$ is monotone as an exercise (see Problem 8.4).
   Let $\{f_i\}_{i\in\mathbb{N}}$ be a chain of functions in $[\mathscr{D} \to \mathscr{E}]$. We will prove that for all $x \in D_\perp$

$$\left(\bigsqcup_{i\in\mathbb{N}} f_i\right)^* (x) = \left(\bigsqcup_{i\in\mathbb{N}} f_i^*\right)(x)$$

if $x = \bot_{D_\bot}$ both sides of the equation simplify to $\bot_E$. So let us assume $x = \lfloor d \rfloor$ for some $d \in D$. We have

$$\begin{aligned}
\left(\bigsqcup_{i\in\mathbb{N}} f_i\right)^* (\lfloor d \rfloor) &= \left(\bigsqcup_{i\in\mathbb{N}} f_i\right)(d) && \text{by definition of lifting} \\
&= \bigsqcup_{i\in\mathbb{N}} (f_i(d)) && \text{by def. of lub in a functional domain} \\
&= \bigsqcup_{i\in\mathbb{N}} (f_i^*(\lfloor d \rfloor)) && \text{by definition of lifting} \\
&= \left(\bigsqcup_{i\in\mathbb{N}} f_i^*\right)(\lfloor d \rfloor) && \text{by def. of lub in a functional domain}
\end{aligned}$$

□

8.5 Continuity Theorems

In this section we show some theorems which allow us to prove the continuity of some functions. We start by proving that the composition of two continuous functions is continuous.

Theorem 8.5 (Continuity of composition). *Let $f \in [D \to E]$ and $g \in [E \to F]$. Their composition*

$$f;g = g \circ f \stackrel{\text{def}}{=} \lambda d.\ g(f(d)) : D \to F$$

is a continuous function, i.e., $g \circ f \in [D \to F]$.

Proof. The statement is just a rephrasing of Theorem 5.5. □

Now we consider a function whose outcome is a pair of values. So the function has a single CPO as domain but it returns a result over a product of CPOs:

$$f : D \to E_1 \times E_2$$

For this type of function we introduce a theorem which allows us to prove the continuity of f in a convenient way. We will consider f as the pairing of two simpler functions $g_1 : D \to E_1$ and $g_2 : D \to E_2$, such that $f(d) = (g_1(d), g_2(d))$ for any $d \in D$. Then we can prove the continuity of f from the continuity of g_1 and g_2 (and vice versa).

Theorem 8.6. *Let $f : D \to E_1 \times E_2$ be a function over CPOs and let*

$$g_1 \stackrel{\text{def}}{=} f;\pi_1 : D \to E_1 \qquad g_2 \stackrel{\text{def}}{=} f;\pi_2 : D \to E_2$$

where $f;\pi_i = \lambda x.\ \pi_i(f(x))$ is the composition of f and π_i for $i = 1,2$. Then f is continuous if and only if g_1 and g_2 are continuous.

Proof. We prove the two implications separately.

$\Rightarrow$) Since f is continuous, by Theorem 8.5 (continuity of composition) and Theorem 8.2 (continuity of projections), also g_1 and g_2 are continuous.

$\Leftarrow$) Note that $\forall d \in D.\ f(d) = (g_1(d), g_2(d))$. We assume the continuity of g_1 and g_2 and we want to prove that f is continuous. Let $\{d_i\}_{i\in\mathbb{N}}$ be a chain in D. We want to prove $f(\bigsqcup_{i\in\mathbb{N}} d_i) = \bigsqcup_{i\in\mathbb{N}} f(d_i)$. So we have

$$\begin{aligned}
f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) &= \left(g_1\left(\bigsqcup_{i\in\mathbb{N}} d_i\right), g_2\left(\bigsqcup_{i\in\mathbb{N}} d_i\right)\right) && \text{(by definition of } g_1, g_2\text{)}\\
&= \left(\bigsqcup_{i\in\mathbb{N}} g_1(d_i), \bigsqcup_{i\in\mathbb{N}} g_2(d_i)\right) && \text{(by continuity of } g_1 \text{ and } g_2\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} (g_1(d_i), g_2(d_i)) && \text{(by definition of lub of pairs)}\\
&= \bigsqcup_{i\in\mathbb{N}} f(d_i) && \text{(by definition of } g_1, g_2\text{)}
\end{aligned}$$

□

Now let us consider the case of a function $f : D_1 \times D_2 \to E$ over CPOs which takes a pair of arguments in D_1 and D_2 and returns an element of E. The following theorem allows us to study the continuity of f by analysing each parameter separately.

Theorem 8.7. *Let $f : D_1 \times D_2 \to E$ be a function over CPOs. Then f is continuous if and only if all the functions in the following two classes are continuous:*

1. $\forall d' \in D_1.\ f_{d'} : D_2 \to E$ *is defined as* $f_{d'} \stackrel{\text{def}}{=} \lambda y.f(d', y)$;
2. $\forall d'' \in D_2.\ f_{d''} : D_1 \to E$ *is defined as* $f_{d''} \stackrel{\text{def}}{=} \lambda x.f(x, d'')$.

Proof. We prove the two implications separately:

$\Rightarrow$) If f is continuous then for all $d' \in D_1, d'' \in D_2$ the functions $f_{d'}$ and $f_{d''}$ are continuous, since we are considering only certain chains (where one element of the pair is fixed). For example, let us fix $d' \in D_1$ and consider a chain $\{d''_i\}_{i\in\mathbb{N}}$ in D_2. Then we prove that $f_{d'}$ is continuous as follows:

$$\begin{aligned}
f_{d'}\left(\bigsqcup_{i\in\mathbb{N}} d''_i\right) &= f\left(d', \bigsqcup_{i\in\mathbb{N}} d''_i\right) && \text{(by definition of } f_{d'}\text{)}\\
&= f\left(\bigsqcup_{i\in\mathbb{N}} (d', d''_i)\right) && \text{(by definition of lub)}\\
&= \bigsqcup_{i\in\mathbb{N}} f(d', d''_i) && \text{(by continuity of } f\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} f_{d'}(d''_i) && \text{(by definition of } f_{d'}\text{)}
\end{aligned}$$

Similarly, if we fix $d'' \in D_2$ and take a chain $\{d'_i\}_{i\in\mathbb{N}}$ in D_1 we have $f_{d''}(\bigsqcup_{i\in\mathbb{N}} d'_i) = \bigsqcup_{i\in\mathbb{N}} f_{d''}(d'_i)$.

$\Leftarrow$) In the opposite direction, assume that $f_{d'}$ and $f_{d''}$ are continuous for all elements $d' \in D_1$ and $d'' \in D_2$. We want to prove that f is continuous. Take a chain $\{(d'_k, d''_k)\}_{k\in\mathbb{N}}$. By definition of lub on pairs, we have

$$\bigsqcup_{k\in\mathbb{N}} (d'_k, d''_k) = \left(\bigsqcup_{i\in\mathbb{N}} d'_i, \bigsqcup_{j\in\mathbb{N}} d''_j \right)$$

Let $d'' \stackrel{\text{def}}{=} \bigsqcup_{j\in\mathbb{N}} d''_j$. It follows that

$$\begin{aligned}
f\left(\bigsqcup_{k\in\mathbb{N}} (d'_k, d''_k) \right) &= f\left(\bigsqcup_{i\in\mathbb{N}} d'_i, \bigsqcup_{j\in\mathbb{N}} d''_j \right) && \text{(by definition of lub on pairs)} \\
&= f\left(\bigsqcup_{i\in\mathbb{N}} d'_i, d'' \right) && \text{(by definition of } d'') \\
&= f_{d''}\left(\bigsqcup_{i\in\mathbb{N}} d'_i \right) && \text{(by definition of } f_{d''}) \\
&= \bigsqcup_{i\in\mathbb{N}} f_{d''}(d'_i) && \text{(by continuity of } f_{d''}) \\
&= \bigsqcup_{i\in\mathbb{N}} f(d'_i, d'') && \text{(by definition of } f_{d''}) \\
&= \bigsqcup_{i\in\mathbb{N}} f_{d'_i}(d'') && \text{(by definition of } f_{d'_i}) \\
&= \bigsqcup_{i\in\mathbb{N}} f_{d'_i}\left(\bigsqcup_{j\in\mathbb{N}} d''_j \right) && \text{(by definition of } d'') \\
&= \bigsqcup_{i\in\mathbb{N}} \bigsqcup_{j\in\mathbb{N}} f_{d'_i}(d''_j) && \text{(by continuity of } f_{d'_i}) \\
&= \bigsqcup_{i\in\mathbb{N}} \bigsqcup_{j\in\mathbb{N}} f(d'_i, d''_j) && \text{(by definition of } f_{d'_i}) \\
&= \bigsqcup_{k\in\mathbb{N}} f(d'_k, d''_k) && \text{(by Lemma 8.1 (switch lemma))}
\end{aligned}$$

□

8.6 Apply, Curry and Fix

As we have done for IMP we will use the lambda notation as a meta-language for the denotational semantics of HOFL. In Section 8.2 we have already defined two new continuous functions for our meta-language (π_1 and π_2). In this section we introduce some additional functions that will form the kernel of our meta-language.

Definition 8.7 (Apply). Let D and E be two CPOs. We define a function apply : $[D \to E] \times D \to E$ as follows:

$$\text{apply}(f,d) \stackrel{\text{def}}{=} f(d)$$

The function apply represents the application of a function in our meta-language: it takes a continuous function $f : D \to E$ and an element $d \in D$ and then returns $f(d)$ as a result. We leave it as an exercise to prove that apply is monotone (see Problem 8.5). We prove that it is also continuous.

Theorem 8.8 (Continuity of apply). *Let* apply : $[D \to E] \times D \to E$ *be the function defined above and let* $\{(f_n,d_n)\}_{n\in\mathbb{N}}$ *be a chain in the CPO*$_\perp$ $[\mathscr{D} \to \mathscr{E}] \times \mathscr{D}$. *Then*

$$\text{apply}\left(\bigsqcup_{n\in\mathbb{N}} (f_n,d_n)\right) = \bigsqcup_{n\in\mathbb{N}} \text{apply}(f_n,d_n)$$

Proof. By Theorem 8.7 we can prove continuity on each parameter separately.

- Let us fix $d \in D$ and take a chain $\{f_n\}_{n\in\mathbb{N}}$ in $[D \to E]$. We have

$$\begin{aligned}
\text{apply}\left(\left(\bigsqcup_{n\in\mathbb{N}} f_n\right),d\right) &= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)(d) && \text{(by definition)}\\
&= \bigsqcup_{n\in\mathbb{N}} (f_n(d)) && \text{(by definition of lub of functions)}\\
&= \bigsqcup_{n\in\mathbb{N}} \text{apply}(f_n,d) && \text{(by definition)}
\end{aligned}$$

- Now we fix $f \in [D \to E]$ and take a chain $\{d_n\}_{n\in\mathbb{N}}$ in D. We have

$$\begin{aligned}
\text{apply}\left(f, \bigsqcup_{n\in\mathbb{N}} d_n\right) &= f\left(\bigsqcup_{n\in\mathbb{N}} d_n\right) && \text{(by definition)}\\
&= \bigsqcup_{n\in\mathbb{N}} f(d_n) && \text{(by continuity of } f)\\
&= \bigsqcup_{n\in\mathbb{N}} \text{apply}(f,d_n) && \text{(by definition)}
\end{aligned}$$

So apply is a continuous function. □

Currying is the name of a technique for transforming a function that takes a pair (or, more generally, a tuple) of arguments into a function that takes each argument separately but computes the same result.

Definition 8.8 (Curry and un-curry). We define the function

$$\text{curry} : (D \times E \to F) \to (D \to E \to F)$$

by letting, for any $g : D \times E \to F$, $d \in D$ and $e \in E$

$$\text{curry } g\ d\ e \stackrel{\text{def}}{=} g(d,e)$$

and we define the function

$$\text{un-curry} : (D \to E \to F) \to (D \times E \to F)$$

by letting, for any $h : D \to E \to F$, $d \in D$ and $e \in E$

$$\text{un-curry } h\ (d,e) \stackrel{\text{def}}{=} h\ d\ e$$

Theorem 8.9 (Continuity of curry). *Let D, E, F be CPOs and $g : D \times E \to F$ be a continuous function. Then $(\text{curry } g) : D \to (E \to F)$ is a continuous function, namely given any chain $\{d_i\}_{i \in \mathbb{N}}$ in D*

$$(\text{curry } g)\left(\bigsqcup_{i \in \mathbb{N}} d_i\right) = \bigsqcup_{i \in \mathbb{N}} (\text{curry } g)(d_i).$$

Proof. Let us note that since g is continuous, by Theorem 8.7, g is continuous separately on each argument. Then let us take $e \in E$. We have

$$\begin{aligned}
(\text{curry } g)\left(\bigsqcup_{i \in \mathbb{N}} d_i\right)(e) &= g\left(\left(\bigsqcup_{i \in \mathbb{N}} d_i\right), e\right) && \text{(by definition of curry } g\text{)} \\
&= \bigsqcup_{i \in \mathbb{N}} g(d_i, e) && \text{(by continuity of } g\text{)} \\
&= \bigsqcup_{i \in \mathbb{N}} ((\text{curry } g)(d_i)(e)) && \text{(by definition of curry } g\text{)}
\end{aligned}$$

□

To define the denotational semantics of recursive definitions we need to provide a fixpoint operator. So it seems useful to introduce fix in our meta-language.

Definition 8.9 (Fix). Let D be a $CPO_\perp$. We define $\text{fix} : [D \to D] \to D$ as

$$\text{fix} \stackrel{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} \lambda f.f^i(\perp_D)$$

Note that, since $\{\lambda f.\ f^i(\bot_D)\}_{i\in\mathbb{N}}$ is a chain of functions and $[D \to D] \to D$ is complete, we are guaranteed that the lub $\bigsqcup_{i\in\mathbb{N}} \lambda f.f^i(\bot_D)$ exists.

Theorem 8.10 (Continuity of fix). *The function* $\mathrm{fix} : [D \to D] \to D$ *is continuous, namely* $\mathrm{fix} \in [[D \to D] \to D]$.

Proof. We know that $[[D \to D] \to D]$ is complete, thus if for all $i \in \mathbb{N}$ the function $\lambda f.\ f^i(\bot_D)$ is continuous, then $\mathrm{fix} = \bigsqcup_{i\in\mathbb{N}} \lambda f.\ f^i(\bot_D)$ is also continuous. We prove that $\forall i \in \mathbb{N}.\ \lambda f.\ f^i(\bot_D)$ is continuous by mathematical induction on i.

Base case: $\lambda f.\ f^0(\bot_D) = \lambda f.\ \bot_D$ is a constant, and thus continuous, function.

Inductive case: Let us assume that $g \stackrel{\text{def}}{=} \lambda f.\ f^i(\bot_D)$ is continuous, i.e., that given a chain $\{f_n\}_{n\in\mathbb{N}}$ in $[D \to D]$ we have $g(\bigsqcup_{n\in\mathbb{N}} f_n) = \bigsqcup_{n\in\mathbb{N}} g(f_n)$, and let us prove that $h \stackrel{\text{def}}{=} \lambda f.\ f^{i+1}(\bot_D)$ is continuous, namely that $h(\bigsqcup_{n\in\mathbb{N}} f_n) = \bigsqcup_{n\in\mathbb{N}} h(f_n)$. In fact we have

$$
\begin{aligned}
h\left(\bigsqcup_{n\in\mathbb{N}} f_n\right) &= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)^{i+1}(\bot_D) && \text{(by def. of } h\text{)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(\left(\bigsqcup_{n\in\mathbb{N}} f_n\right)^{i}(\bot_D)\right) && \text{(by def. of } (\cdot)^{i+1}\text{)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(g\left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\right) && \text{(by def. of } g\text{)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(\bigsqcup_{n\in\mathbb{N}} g(f_n)\right) && \text{(by ind. hyp.)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(\bigsqcup_{n\in\mathbb{N}} f_n^i(\bot_D)\right) && \text{(by def of } g\text{)}\\
&= \bigsqcup_{n\in\mathbb{N}}\left(f_n\left(\bigsqcup_{m\in\mathbb{N}} f_m^i(\bot_D)\right)\right) && \text{(by def. of lub)}\\
&= \bigsqcup_{n\in\mathbb{N}}\bigsqcup_{m\in\mathbb{N}} f_n\left(f_m^i(\bot_D)\right) && \text{(by cont. of } f_n\text{)}\\
&= \bigsqcup_{k\in\mathbb{N}} f_k\left(f_k^i(\bot_D)\right) && \text{(by Lemma 8.1)}\\
&= \bigsqcup_{k\in\mathbb{N}} f_k^{i+1}(\bot_D) && \text{(by def. of } (\cdot)^{i+1}\text{)}\\
&= \bigsqcup_{n\in\mathbb{N}} h(f_n) && \text{(by def. of } h\text{)}
\end{aligned}
$$

□

Finally we introduce the **let** operator, whose role is that of binding a name x to a de-lifted expression. Note that the continuity of the **let** operator directly follows from the continuity of the lifting operator.

Definition 8.10 (Let operator). Let $\mathscr{E}$ be a $CPO_\perp$ and $\lambda x.\ e$ a function in $[D \to E]$. We define the **let** operator as follows, where $d' \in D_\perp$:

$$\textbf{let}\ x \Leftarrow d'.\ e \stackrel{\text{def}}{=} \underbrace{\underbrace{\underbrace{(\lambda x.\ e)}_{D\to E}{}^{*}}_{D_\perp \to E}\ \underbrace{(d')}_{D_\perp}}_{E} = \begin{cases} \perp_E & \text{if } d' = \perp_{D_\perp} \\ e\left[{}^{d}/_{x}\right] & \text{if } d' = \lfloor d \rfloor \text{ for some } d \in D \end{cases}$$

Intuitively, given $d' \in D_\perp$, if $d' = \perp$ then **let** $x \Leftarrow d'.\ e$ returns $\perp_E$, otherwise it means that $d' = \lfloor d \rfloor$ for some $d \in D$ and thus it returns $e\left[{}^{d}/_{x}\right]$, as if $\lambda x.\ e$ was applied to d, i.e., $d' = \lfloor d \rfloor$ is de-lifted so that $\lambda x.\ e$ can be used.

Problems

8.1. Prove that the projection functions in Definition 8.3 are monotone.

8.2. Prove that the domain $[\mathscr{D} \to \mathscr{E}]$ from Definition 8.4 is a $\text{CPO}_\perp$.

8.3. Prove that the lifted domain $\mathscr{D}_\perp$ from Definition 8.5 is a $\text{CPO}_\perp$.

8.4. Complete the part of the proof of Theorem 8.4 concerned with the monotonicity of the lifting function $(\cdot)^*$.

8.5. Prove that the function apply : $[D \to E] \times D \to E$ introduced in Definition 8.7 is monotone.

8.6. Let D be a CPO and $f : D \to D$ be a continuous function. Prove that the set of fixpoints of f is itself a CPO (under the order inherited from D).

8.7. Let D and E be two $\text{CPO}_\perp$s. A function $f : D \to E$ is called *strict* if $f(\perp_D) = \perp_E$. Prove that the set of strict functions from D to E is a $\text{CPO}_\perp$ under the usual order.

8.8. Let D and E be two CPOs. Prove that the following two definitions of the order between continuous functions $f, g : D \to E$ are equivalent:

1. $f \sqsubseteq g \ \Leftrightarrow\ \forall d \in D.\ f(d) \sqsubseteq_E g(d)$
2. $f \preceq g \ \Leftrightarrow\ \forall d_1, d_2 \in D.\ (d_1 \sqsubseteq_D d_2 \Rightarrow f(d_1) \sqsubseteq_E g(d_2))$

8.9. Let $\mathscr{D} = (D, \sqsubseteq_D)$ and $\mathscr{E} = (E, \sqsubseteq_E)$ be two CPOs. Their sum $\mathscr{D} + \mathscr{E}$ has

1. The set of elements

$$\{\perp\} \uplus D \uplus E = \{(0, \perp)\} \cup (\ \{1\} \times ((\{0\} \times D) \cup (\{1\} \times E))\ )$$

2. The order relation $\sqsubseteq_{D+E}$ defined by letting
   - $(1,(0,d_1)) \sqsubseteq_{D+E} (1,(0,d_2))$ if $d_1 \sqsubseteq_D d_2$
   - $(1,(1,e_1)) \sqsubseteq_{D+E} (1,(1,e_2))$ if $e_1 \sqsubseteq_E e_2$
   - $\forall x \in \{\bot\} \uplus D \uplus E.\ (0,\bot) \sqsubseteq_{D+E} x$

Prove that $\mathscr{D} + \mathscr{E}$ is a $\mathrm{CPO}_\bot$.

8.10. Prove that un-curry is continuous and inverse to curry (see Definition 8.8).

Chapter 9
Denotational Semantics of HOFL

Work out what you want to say before you decide how you want to say it. (Christopher Strachey's first law of logical design)

Abstract In this chapter we exploit the domain theory from Chapter 8 to define the (lazy) denotational semantics of HOFL. For each type τ we introduce a corresponding domain $(V_\tau)_\perp$ which is defined inductively over the structure of τ and such that we can assign an element of the domain $(V_\tau)_\perp$ to each (closed and typable) term t with type τ. Moreover, we introduce the notion of environment, which assigns meanings to variables, and can be exploited to define the denotational semantics of (typable) terms with variables. Interestingly, all constructions we use are continuous, so that we are able to assign meaning also to any (typable) term that is recursively defined. We conclude the chapter by showing some important properties of the denotational semantics; in particular, that it is compositional.

9.1 HOFL Semantic Domains

In order to specify the denotational semantics of a programming language, we have to define, by structural recursion, an interpretation function from each syntactic domain to a semantic domain. In IMP there are three syntactic domains, *Aexp* for arithmetic expressions, *Bexp* for boolean expressions and *Com* for commands. Correspondingly, we have defined three semantic domains and three interpretation functions ($\mathscr{A}[\![\cdot]\!]$, $\mathscr{B}[\![\cdot]\!]$ and $\mathscr{C}[\![\cdot]\!]$). HOFL has a sole syntactic domain (i.e., the set of well-formed terms t) and thus we have only one interpretation function, written $[\![\cdot]\!]$. However, since HOFL terms are typed, the interpretation function is parametric w.r.t. the type τ of t and we have one semantic domain V_τ for each type τ. Actually, we distinguish between V_τ, where we find the meanings of the terms of type τ with canonical forms, and $(V_\tau)_\perp$, where the additional element $\perp_{(V_\tau)_\perp}$ assigns a meaning to all the terms of type τ without a canonical form. Moreover, we will need to handle terms with free variables, as, e.g., when defining the denotational semantics of $\lambda x.\, t$ in terms of the denotational semantics of t (with x possibly in $\mathrm{fv}(t)$). This was not the case for the operational semantics of HOFL, where only closed terms are considered. As terms may contain free variables, we pass to the interpretation function an *environment*

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_9

$$\rho \in Env \stackrel{\text{def}}{=} Var \to \bigcup_{\tau} (V_\tau)_\perp$$

which assigns meaning to variables. For consistency reasons, any environment ρ that we consider must satisfy the condition $\rho(x) \in (V_\tau)_\perp$ whenever $x : \tau$. Thus, we have

$$[\![t : \tau]\!] : Env \to (V_\tau)_\perp$$

The actual semantic domains V_τ and $(V_\tau)_\perp$ are defined by structural recursion on the syntax of types:

$$\begin{array}{rcl@{\qquad}rcl} V_{int} & \stackrel{\text{def}}{=} & \mathbb{Z} & (V_{int})_\perp & \stackrel{\text{def}}{=} & \mathbb{Z}_\perp \\ V_{\tau_1 * \tau_2} & \stackrel{\text{def}}{=} & (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp & (V_{\tau_1 * \tau_2})_\perp & \stackrel{\text{def}}{=} & ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp \\ V_{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp] & (V_{\tau_1 \to \tau_2})_\perp & \stackrel{\text{def}}{=} & [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp \end{array}$$

Notice that the recursive definition above takes advantage of the domain constructors we have defined in Chapter 8. While the lifiting $\mathbb{Z}_\perp$ of the integer numbers $\mathbb{Z}$ is strictly necessary, liftings on cartesian pairs and on continuous functions are actually optional, since cartesian products and functional domains are already $CPO_\perp$s. We will discuss the motivation of our choice at the end of Chapter 10.

9.2 HOFL Interpretation Function

Now we are ready to define the interpretation function, by structural recursion. We briefly comment on each definition and show that the clauses of the structural recursion are typed correctly.

9.2.1 Constants

We define the meaning of a constant as the obvious value on the lifted domain:

$$[\![n]\!]\rho \stackrel{\text{def}}{=} \lfloor n \rfloor$$

At the level of types, we have

$$\underbrace{[\![\underbrace{n}_{int}]\!]\rho}_{(V_{int})_\perp = \mathbb{Z}_\perp} = \underbrace{\lfloor \underbrace{n}_{\mathbb{Z}} \rfloor}_{\mathbb{Z}_\perp}$$

9.2.2 Variables

The meaning of a variable is defined by its value in the given environment ρ:

$$[\![x]\!]\rho \stackrel{\text{def}}{=} \rho(x)$$

It is obvious that the typing is respected (under the assumption that $\rho(x) \in (V_\tau)_\perp$ whenever $x : \tau$):

$$\underbrace{[\![\underbrace{x}_{\tau}]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\rho(\underbrace{x}_{\tau})}_{(V_\tau)_\perp}$$

9.2.3 Arithmetic Operators

We give the generic semantics of a binary operator $\text{op} \in \{+,-,\times\}$ as

$$[\![t_0 \text{ op } t_1]\!]\rho = [\![t_0]\!]\rho \; \underline{\text{op}}_\perp \; [\![t_1]\!]\rho$$

where for any operator $\text{op} \in \{+,-,\times\}$ in the syntax we have the corresponding function $\underline{\text{op}} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ on the integers $\mathbb{Z}$ and also the binary function $\underline{\text{op}}_\perp$ on $\mathbb{Z}_\perp$ defined as

$$\underline{\text{op}}_\perp : (\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$$

$$x_1 \; \underline{\text{op}}_\perp \; x_2 = \begin{cases} \lfloor n_1 \; \underline{op} \; n_2 \rfloor & \text{if } x_1 = \lfloor n_1 \rfloor \text{ and } x_2 = \lfloor n_2 \rfloor \text{ for some } n_1, n_2 \in \mathbb{Z} \\ \perp_{\mathbb{Z}_\perp} & \text{otherwise} \end{cases}$$

We remark that $\underline{\text{op}}_\perp$ yields $\perp_{\mathbb{Z}_\perp}$ when at least one of the two arguments is $\perp_{\mathbb{Z}_\perp}$.

At the level of types, we have

$$\underbrace{[\![\underbrace{(\underbrace{t_0}_{int} \text{ op } \underbrace{t_1}_{int})}_{int}]\!]\rho}_{(V_{int})_\perp = \mathbb{Z}_\perp} = \underbrace{[\![\underbrace{t_0}_{int}]\!]\rho \quad \underbrace{\underline{\text{op}}_\perp}_{(\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \to \mathbb{Z}_\perp} \quad [\![\underbrace{t_1}_{int}]\!]\rho}_{(V_{int})_\perp}$$

9.2.4 Conditional

In order to define the semantics of the conditional expression, we exploit the conditional operator of the meta-language

$$Cond_\tau : \mathbb{Z}_\perp \times (V_\tau)_\perp \times (V_\tau)_\perp \to (V_\tau)_\perp$$

defined as

$$Cond_\tau(v,d_0,d_1) \stackrel{\text{def}}{=} \begin{cases} d_0 & \text{if } v = \lfloor 0 \rfloor \\ d_1 & \text{if } \exists n \in \mathbb{Z}.\ v = \lfloor n \rfloor \wedge n \neq 0 \\ \bot_{(V_\tau)_\bot} & \text{if } v = \bot_{\mathbb{Z}_\bot} \end{cases}$$

Note that $Cond_\tau$ is parametric on the type τ. In the following, when τ can be inferred, we write just $Cond$. The conditional operator is *strict* on its first argument (i.e., it returns $\bot$ when the first argument is $\bot$) but not on the second and third arguments.

We can now define the denotational semantics of the conditional operator by letting

$$[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho \stackrel{\text{def}}{=} Cond\,([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

At the level of types, we have

$$\underbrace{[\![\textbf{if } \underbrace{t_0}_{int} \textbf{ then } \underbrace{t_1}_{\tau} \textbf{ else } \underbrace{t_2}_{\tau}]\!]\rho}_{(V_\tau)_\bot} = \underbrace{\underbrace{Cond_\tau}_{\mathbb{Z}_\bot\times(V_\tau)_\bot\times(V_\tau)_\bot\to(V_\tau)_\bot}\ \underbrace{([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)}_{(V_{int})_\bot\times(V_\tau)_\bot\times(V_\tau)_\bot}}_{(V_\tau)_\bot}$$

9.2.5 Pairing

For the pairing operator we simply let

$$[\![(t_0,t_1)]\!]\rho \stackrel{\text{def}}{=} \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor$$

Note that, for $t_0 : \tau_0$ and $t_1 : \tau_1$, the pair $([\![t_0]\!]\rho, [\![t_1]\!]\rho)$ is in $(V_{\tau_0})_\bot \times (V_{\tau_1})_\bot$ and not in $((V_{\tau_0})_\bot \times (V_{\tau_1})_\bot)_\bot$, thus we apply the lifting. In fact, at the level of type consistency we have

$$\underbrace{[\![(\underbrace{t_0}_{\tau_0}, \underbrace{t_1}_{\tau_1})]\!]\rho}_{(V_{\tau_0*\tau_1})_\bot} = \underbrace{\lfloor ([\![\underbrace{t_0}_{\tau_0}]\!]\rho, [\![\underbrace{t_1}_{\tau_1}]\!]\rho) \rfloor}_{((V_{\tau_0})_\bot\times(V_{\tau_1})_\bot)_\bot}$$

9.2.6 Projections

We define the projections by using the lifted version of the projections π_1 and π_2 of the meta-language:

$$\begin{aligned}[\![\mathbf{fst}(t)]\!]\rho &\stackrel{\text{def}}{=} \mathbf{let}\ d \Leftarrow [\![t]\!]\rho.\ \pi_1\ d \\ &= \pi_1^*([\![t]\!]\rho) \\ [\![\mathbf{snd}(t)]\!]\rho &\stackrel{\text{def}}{=} \mathbf{let}\ d \Leftarrow [\![t]\!]\rho.\ \pi_2\ d \\ &= \pi_2^*([\![t]\!]\rho)\end{aligned}$$

The **let** operator (see Definition 8.10) allows us to *de-lift* $[\![t]\!]\rho$ in order to apply projections π_1 and π_2. Instead, if $[\![t]\!]\rho = \bot$ the result is also $\bot$.

Again, we check that the type constraints are respected by the definition:

$$\underbrace{[\![\mathbf{fst}(\underbrace{\underbrace{t}_{\tau_0 * \tau_1}}_{\tau_0})]\!]\rho}_{(V_{\tau_0})_\bot} = \mathbf{let}\ \underbrace{d}_{V_{\tau_0*\tau_1}} \Leftarrow \underbrace{[\![\ \underbrace{t}_{\tau_0*\tau_1}\ ]\!]\rho}_{(V_{\tau_0*\tau_1})_\bot}.\ \underbrace{\underbrace{\pi_1}_{(V_{\tau_0})_\bot \times (V_{\tau_1})_\bot \to (V_{\tau_0})_\bot}\ \underbrace{d}_{V_{\tau_0*\tau_1}}}_{(V_{\tau_0})_\bot}$$

The case of $\mathbf{snd}(t : \tau_0 * \tau_1)$ is completely analogous and thus omitted.

9.2.7 Lambda Abstraction

For lambda abstraction we use, of course, the lambda operator of the meta-language:

$$[\![\lambda x.\ t]\!]\rho \stackrel{\text{def}}{=} \left\lfloor \lambda d.\ [\![t]\!]\rho[^d/_x] \right\rfloor$$

where we bind x to d for evaluating t.

Note that, as in the case of pairing, we need to apply the lifting, because $\lambda d.\ [\![t]\!]\rho[^d/_x]$ is an element of $V_{\tau_0\to\tau_1} = [(V_{\tau_0})_\bot \to (V_{\tau_1})_\bot]$ and not of $(V_{\tau_0\to\tau_1})_\bot = [(V_{\tau_0})_\bot \to (V_{\tau_1})_\bot]_\bot$:

$$\underbrace{[\![\lambda\ \underbrace{\underbrace{x}_{\tau_0}.\ \underbrace{t}_{\tau_1}}_{\tau_0\to\tau_1}]\!]\rho}_{(V_{\tau_0\to\tau_1})_\bot} = \underbrace{\underbrace{\lfloor\lambda\ \underbrace{d}_{(V_{\tau_0})_\bot}\ .\ \underbrace{[\![\underbrace{t}_{\tau_1}]\!]\rho[^d/_x]}_{(V_{\tau_1})_\bot}\rfloor}_{[(V_{\tau_0})_\bot\to(V_{\tau_1})_\bot]}}_{[(V_{\tau_0})_\bot\to(V_{\tau_1})_\bot]_\bot}$$

9.2.8 Function Application

Similarly to the case of projections, we apply the de-lifted version of the function to its argument:

$$\begin{aligned}[\![(t_1\ t_0)]\!]\rho &\stackrel{\text{def}}{=} \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho)\\ &= (\lambda\varphi.\ \varphi([\![t_0]\!]\rho))^*([\![t_1]\!]\rho)\end{aligned}$$

At the level of types, we have:

$$\underbrace{[\![(\underbrace{\underbrace{t_1}_{\tau_0\to\tau_1}\ \underbrace{t_0}_{\tau_0}}_{\tau_1})]\!]\rho}_{(V_{\tau_1})_\perp} = \underbrace{\textbf{let}\ \underbrace{\varphi}_{V_{\tau_0\to\tau_1}} \Leftarrow \underbrace{[\![t_1]\!]\rho}_{(V_{\tau_0\to\tau_1})_\perp}.\ \underbrace{\underbrace{\varphi}_{V_{\tau_0\to\tau_1}}\ \underbrace{([\![t_0]\!]\rho)}_{(V_{\tau_0})_\perp}}_{(V_{\tau_1})_\perp}}_{(V_{\tau_1})_\perp}$$

9.2.9 Recursion

For handling recursion we would like to find a solution (in the domain $(V_\tau)_\perp$, for $t : \tau$) to the recursive equation

$$[\![\textbf{rec}\ x.\ t]\!]\rho = [\![t]\!]\rho[{}^{[\![\textbf{rec}\ x.\ t]\!]\rho}/_x]$$

The least solution can be computed simply by applying the fix operator of the meta-language:

$$[\![\textbf{rec}\ x.\ t]\!]\rho \stackrel{\text{def}}{=} \text{fix}\ \lambda d.\ [\![t]\!]\rho[{}^d/_x]$$

Finally, we check that also this last definition is consistent with the typing:

$$\underbrace{[\![\textbf{rec}\ \underbrace{x}_{\tau}.\ \underbrace{t}_{\tau}]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\underbrace{\text{fix}}_{[[(V_\tau)_\perp\to(V_\tau)_\perp]\to(V_\tau)_\perp]}\ \underbrace{\lambda\ \underbrace{d}_{(V_\tau)_\perp}.\ \underbrace{[\![\underbrace{t}_{\tau}]\!]\rho[{}^d/_x]}_{(V_\tau)_\perp}}_{[(V_\tau)_\perp\to(V_\tau)_\perp]}}_{(V_\tau)_\perp}$$

9.2.10 Eager Semantics

The denotational semantics we have defined is *lazy*, in the sense that the evaluation of the argument is not enforced by the interpretation of application. The corresponding *eager* variant could be defined simply by letting

$$[\![(t_1\ t_0)]\!]\rho \stackrel{\text{def}}{=} \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \textbf{let}\ d \Leftarrow [\![t_0]\!]\rho.\ \varphi(d)$$

The difference is that, according to the eager semantics, $[\![(t_1\ t_0)]\!]\rho$ evaluates to $\perp$ when $[\![t_0]\!]\rho$ evaluates to $\perp$, while this is not necessarily the case in the lazy semantics.

9.2.11 Examples

Example 9.1. Let us see some simple examples of evaluation of the denotational semantics. We consider three similar terms f, g and h such that f and h have the same denotational semantics while g has a different semantics because it requires a parameter x to be evaluated even if it is not used:

1. $f \stackrel{\text{def}}{=} \lambda x : int.\ 3$
2. $g \stackrel{\text{def}}{=} \lambda x : int.\ \textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3$
3. $h \stackrel{\text{def}}{=} \textbf{rec}\, y : int \to int.\ \lambda x : int.\ 3$

Note that $f, g, h : int \to int$. For the term f we have

$$[\![f]\!]\rho = [\![\lambda x.\ 3]\!]\rho = \lfloor \lambda d.\ [\![3]\!]\rho[^d/_x] \rfloor = \lfloor \lambda d.\ \lfloor 3 \rfloor \rfloor$$

When considering g, instead

$$\begin{aligned} [\![g]\!]\rho &= [\![\lambda x.\ \textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3]\!]\rho \\ &= \lfloor \lambda d.\ [\![\textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3]\!]\rho[^d/_x] \rfloor \\ &= \lfloor \lambda d.\ Cond(d, \lfloor 3 \rfloor, \lfloor 3 \rfloor) \rfloor \\ &= \lfloor \lambda d.\ \textbf{let}\ x \Leftarrow d.\ \lfloor 3 \rfloor \rfloor \end{aligned}$$

where the last equality follows from the fact that both expressions $Cond(d, \lfloor 3 \rfloor, \lfloor 3 \rfloor)$ and $\textbf{let}\ x \Leftarrow d.\ \lfloor 3 \rfloor$ evaluate to $\bot_{\mathbb{Z}_\bot}$ when $d = \bot_{\mathbb{Z}_\bot}$ and to $\lfloor 3 \rfloor$ if d is a lifted value. Thus we can conclude that $[\![f]\!]\rho \neq [\![g]\!]\rho$.

Finally, for h we get

$$\begin{aligned} [\![h]\!]\rho &= [\![\textbf{rec}\ y.\ \lambda x.\ 3]\!]\rho \\ &= \text{fix}\ \lambda d_y.\ [\![\lambda x.\ 3]\!]\rho[^{d_y}/_y] \\ &= \text{fix}\ \lambda d_y.\ \lfloor \lambda d_x.\ [\![3]\!]\rho[^{d_y}/_y, ^{d_x}/_x] \rfloor \\ &= \text{fix}\ \lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor \end{aligned}$$

Let $\Gamma_h = \lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor$. We can compute the fixpoint by exploiting the fixpoint theorem to compute successive approximations:

$$\begin{aligned} d_0 &= \Gamma_h^0(\bot_{[\mathbb{Z}_\bot \to \mathbb{Z}_\bot]_\bot}) = \bot_{[\mathbb{Z}_\bot \to \mathbb{Z}_\bot]_\bot} \\ d_1 &= \Gamma_h(d_0) = (\lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor)\bot = \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor \\ d_2 &= \Gamma_h(d_1) = (\lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor)\lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor = \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor = d_1 \end{aligned}$$

Since $d_2 = d_1$ we have reached the fixpoint and thus

$$[\![h]\!]\rho = \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor = [\![f]\!]\rho.$$

Note that we could have avoided the calculation of d_2, because d_1 is already a maximal element in $[\mathbb{Z}_\bot \to \mathbb{Z}_\bot]_\bot$ and therefore it must be that $\Gamma_h(d_1) = d_1$.

9.3 Continuity of Meta-language's Functions

In order to show that the semantics is always well defined we have to show that all the functions we employ in the definition are continuous, so that the fixpoint theory is applicable.

Theorem 9.1. *The following functions are monotone and continuous:*

1. $\underline{\mathrm{op}}_{\perp} : (\mathbb{Z}_{\perp} \times \mathbb{Z}_{\perp}) \to \mathbb{Z}_{\perp}$;
2. $Cond_{\tau} : \mathbb{Z}_{\perp} \times (V_{\tau})_{\perp} \times (V_{\tau})_{\perp} \to (V_{\tau})_{\perp}$;
3. $(\_,\_) : (V_{\tau_0})_{\perp} \times (V_{\tau_1})_{\perp} \to V_{\tau_0 * \tau_1}$;
4. $\pi_1 : V_{\tau_0 * \tau_1} \to (V_{\tau_0})_{\perp}$;
5. $\pi_2 : V_{\tau_0 * \tau_1} \to (V_{\tau_1})_{\perp}$;
6. **let**;
7. apply;
8. $\mathrm{fix} : [[(V_{\tau})_{\perp} \to (V_{\tau})_{\perp}] \to (V_{\tau})_{\perp}]$.

Proof. Monotonocity is obvious in most cases. We focus on the continuity of the various functions:

1. Since $\underline{\mathrm{op}}_{\perp}$ is monotone over a domain with only finite chains then it is also continuous.
2. By using Theorem 8.7, we can prove the continuity of *Cond* on each parameter separately.
   Let us show the continuity on the first parameter. Since chains in $\mathbb{Z}_{\perp}$ are finite, it is enough to prove monotonicity. We fix $d_1, d_2 \in (V_{\tau})_{\perp}$ and we prove the monotonicity of $\lambda x.\, Cond_{\tau}(x, d_1, d_2) : \mathbb{Z}_{\perp} \to (V_{\tau})_{\perp}$. Let $n, m \in \mathbb{Z}$:

   - the cases $\perp_{\mathbb{Z}_{\perp}} \sqsubseteq_{\mathbb{Z}_{\perp}} \perp_{\mathbb{Z}_{\perp}}$ and $\lfloor n \rfloor \sqsubseteq_{\mathbb{Z}_{\perp}} \lfloor n \rfloor$ are trivial;
   - for the case $\perp_{\mathbb{Z}_{\perp}} \sqsubseteq_{\mathbb{Z}_{\perp}} \lfloor n \rfloor$ obviously

     $$Cond_{\tau}(\perp_{\mathbb{Z}_{\perp}}, d_1, d_2) = \perp_{(V_{\tau})_{\perp}} \sqsubseteq_{(V_{\tau})_{\perp}} Cond_{\tau}(\lfloor n \rfloor, d_1, d_2)$$

     because $\perp_{(V_{\tau})_{\perp}}$ is the bottom element of $(V_{\tau})_{\perp}$;
   - for the case $\lfloor n \rfloor \sqsubseteq_{\mathbb{Z}_{\perp}} \lfloor m \rfloor$, since $\mathbb{Z}_{\perp}$ is a flat domain we have $n = m$ and trivially $Cond_{\tau}(\lfloor n \rfloor, d_1, d_2) \sqsubseteq_{(V_{\tau})_{\perp}} Cond_{\tau}(\lfloor m \rfloor, d_1, d_2)$.

   Now let us show the continuity on the second parameter, namely we fix $v \in \mathbb{Z}_{\perp}$ and $d \in (V_{\tau})_{\perp}$ and for any chain $\{d_i\}_{i \in \mathbb{N}}$ in $(V_{\tau})_{\perp}$ we prove that

   $$Cond_{\tau}\left(v, \bigsqcup_{i \in \mathbb{N}} d_i, d\right) = \bigsqcup_{i \in \mathbb{N}} Cond_{\tau}(v, d_i, d)$$

   - if $v = \perp_{\mathbb{Z}_{\perp}}$, then

     $$Cond_{\tau}\left(\perp_{\mathbb{Z}_{\perp}}, \bigsqcup_{i \in \mathbb{N}} d_i, d\right) = \perp_{(V_{\tau})_{\perp}} = \bigsqcup_{i \in \mathbb{N}} \perp_{(V_{\tau})_{\perp}} = \bigsqcup_{i \in \mathbb{N}} Cond_{\tau}(\perp_{\mathbb{Z}_{\perp}}, d_i, d)$$

- if $v = \lfloor 0 \rfloor$, then $\lambda x.\ Cond_\tau(\lfloor 0 \rfloor, x, d)$ is the identity function $\lambda x.\ x$ and we have

$$Cond_\tau\left(\lfloor 0 \rfloor, \bigsqcup_{i\in\mathbb{N}} d_i, d\right) = \bigsqcup_{i\in\mathbb{N}} d_i = \bigsqcup_{i\in\mathbb{N}} Cond_\tau(\lfloor 0 \rfloor, d_i, d)$$

- if $v = \lfloor n \rfloor$ with $n \neq 0$, then $\lambda x.\ Cond_\tau(\lfloor n \rfloor, x, d)$ is the constant function $\lambda x.\ d$ and we have

$$Cond_\tau\left(\lfloor n \rfloor, \bigsqcup_{i\in\mathbb{N}} d_i, d\right) = d = \bigsqcup_{i\in\mathbb{N}} d = \bigsqcup_{i\in\mathbb{N}} Cond_\tau(\lfloor n \rfloor, d_i, d)$$

In all cases $Cond_\tau$ is continuous.
Continuity on the third parameter is analogous.

3. For pairing $(\_,\_)$ we can again use Theorem 8.7, which allows us to show separately the continuity on each parameter. If we fix the first element we have

$$\left(d, \bigsqcup_{i\in\mathbb{N}} d_i\right) = \left(\bigsqcup_{i\in\mathbb{N}} d, \bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} (d, d_i)$$

by definition of lub of a chain of pairs (see Theorem 8.1). The same holds for the second parameter.
4. Projections π_1 and π_2 are continuous by Theorem 8.2.
5. The **let** function is continuous since $(\cdot)^*$ is continuous by Theorem 8.4.
6. apply is continuous by Theorem 8.8
7. fix is continuous by Theorem 8.10. □

In the previous theorem we have not mentioned the continuity proofs for lambda abstraction and recursion. The next theorem fills these gaps.

Theorem 9.2. *Let $t : \tau$ be a well-typed term of HOFL; then the following holds:*

1. *$(\lambda d.\ [\![t]\!]\rho[^d/_x])$ is a continuous function;*
2. *if $\tau = \tau_0 \to \tau_1$ is a functional type, then* fix *$\lambda d.\ [\![t]\!]\rho[^d/_x]$ is a continuous function.*

Proof. Let us prove the two properties.

1. We prove the stronger property that, for any $n \in \mathbb{N}$

$$\lambda(d_1, ..., d_n).\ [\![t]\!]\rho[^{d_1}/_{x_1}, \cdots, ^{d_n}/_{x_n}]$$

is a continuous function. The proof is by structural induction on t. Below, for brevity, we write $\widetilde{d}$ instead of $d_1, ..., d_n$ and ρ' instead of $\rho[^{d_1}/_{x_1}, \cdots, ^{d_n}/_{x_n}]$:

$t = y$: Then $\lambda\widetilde{d}.\ [\![y]\!]\rho'$ is either a projection function (if $y = x_i$ for some $i \in [1,n]$) or the constant function $\lambda\widetilde{d}.\ \rho(y)$ (if $y \notin \{x_1, ..., x_n\}$), which are continuous.

$t = t_1 \text{ op } t_2$: By the inductive hypothesis $f_1 \stackrel{\text{def}}{=} \lambda\widetilde{d}.\ [\![t_1]\!]\rho'$ and $f_2 \stackrel{\text{def}}{=} \lambda\widetilde{d}.\ [\![t_2]\!]\rho'$ are continuous. Then $f \stackrel{\text{def}}{=} \lambda\widetilde{d}.\ ((f_1\ \widetilde{d}),(f_2\ \widetilde{d}))$ is continuous, and

$$\begin{aligned}\lambda\widetilde{d}.\ [\![t_1 \text{ op } t_2]\!]\rho' &= \lambda\widetilde{d}.\ ([\![t_1]\!]\rho'\ \underline{\text{op}}_\perp\ [\![t_2]\!]\rho')\\ &= \lambda\widetilde{d}.\ (f_1\ \widetilde{d})\ \underline{\text{op}}_\perp\ (f_2\ \widetilde{d})\\ &= \underline{\text{op}}_\perp \circ f\end{aligned}$$

is continuous because $\underline{\text{op}}_\perp$ is continuous and the composition of continuous functions yields a continuous function by Theorem 8.5.

$t = \lambda y.\ t'$: By the inductive hypothesis we can assume that $\lambda(\widetilde{d},d).[\![t']\!]\rho'[^d/_y]$ is continuous. Then $\text{curry}(\lambda(\widetilde{d},d).[\![t']\!]\rho'[^d/_y])$ is continuous since curry is continuous, and we conclude by noting that

$$\begin{aligned}\text{curry}(\lambda(\widetilde{d},d).[\![t']\!]\rho'[^d/_y]) &= \lambda\widetilde{d}.\ \lambda d.\ [\![t']\!]\rho'[^d/_y]\\ &= \lambda\widetilde{d}.[\![\lambda y.\ t']\!]\rho'\end{aligned}$$

We leave the remaining cases as an exercise.

2. To prove the second proposition we note that

$$\text{fix}\ \ \lambda d.[\![t]\!]\rho[^d/_x]$$

is the application of a continuous function (i.e., the function fix, by Theorem 8.10) to a continuous argument (i.e., $\lambda d.[\![t]\!]\rho[^d/_x]$, continuous by the first part of this theorem) so it is continuous by Theorem 8.8. □

We conclude this section by recalling that the definition of denotational semantics is consistent with the typing.

Theorem 9.3 (Type consistency). *If $t : \tau$ then $\forall \rho \in Env.\ [\![t]\!]\,\rho \in (V_\tau)_\perp$.*

Proof. The proof is by structural induction on t and it has been outlined when giving the structurally recursive definition of the denotational semantics (where we have also relied on the previous continuity theorems). □

9.4 Substitution Lemma and Other Properties

We conclude this chapter by stating some useful theorems. The most important is the *Substitution Lemma* which states that the substitution operator commutes with the interpretation function.

Theorem 9.4 (Substitution lemma). *Let $x,t : \tau$ and $t' : \tau'$. We have*

$$[\![t'[^t/_x]]\!]\,\rho = [\![t']\!]\,\rho[^{[\![t]\!]\rho}/_x]$$

Proof. By Theorem 7.1 we know that $t'[{}^{t}/_{x}] : \tau'$. The proof is by structural induction on t' and is left as an exercise (see Problem 9.13). □

In words, replacing a variable x with a term t in a term t' returns a term $t'[{}^{t}/_{x}]$ whose denotational semantics $[\![t'[{}^{t}/_{x}]]\!]\,\rho = [\![t']\!]\,\rho[{}^{[\![t]\!]\rho}/_{x}]$ depends only on the denotational semantics $[\![t]\!]\,\rho$ of t.

Remark 9.1 (Compositionality). The substitution lemma is an important result, as it implies the compositionality of denotational semantics, namely for all terms t_1, t_2 and environment ρ we have

$$[\![t_1]\!]\,\rho = [\![t_2]\!]\,\rho \quad \Rightarrow \quad [\![t[{}^{t_1}/_{x}]]\!]\,\rho = [\![t[{}^{t_2}/_{x}]]\!]\,\rho$$

Theorem 9.5. *Let t be a well-defined term of HOFL. Let $\rho, \rho' \in Env$ such that $\forall\, x \in \mathrm{fv}(t).\ \rho(x) = \rho'(x)$ then*

$$[\![t]\!]\,\rho = [\![t]\!]\,\rho'$$

Proof. The proof is by structural induction on t and is left as an exercise (see Problem 9.16). □

Theorem 9.6. *Let $c \in C_\tau$ be a closed term in canonical form of type τ. Then we have*

$$\forall\, \rho \in Env.\ [\![c]\!]\,\rho \neq \bot_{(V_\tau)_\bot}$$

Proof. Immediate, by inspection of the clauses for terms in canonical form. □

Problems

9.1. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ (f(x) \times f(x))$$

Derive the type, the canonical form and the denotational semantics of t.

9.2. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \lambda y.\ \textbf{if}\ x \times y\ \textbf{then}\ x\ \textbf{else}\ (fx)((fx)y)$$

Derive the type, the canonical form and the denotational semantics of t.

9.3. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{fst}(\ (\lambda x.\ x)\ (\ 1\ ,\ ((\textbf{rec}\, f.\ \lambda y.\ (f\ y))\ 2)\ )\ ).$$

Derive the type, the canonical form and the denotational semantics of t.

9.4. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (g\ (f\ (x-1)))$$

1. Derive the type of t and the denotational semantics of $[\![t]\!]\rho$ by assuming that $\rho g = \lfloor h \rfloor$ for some suitable h.
2. Compute the canonical form of the term $(((\lambda g.\, t)\ \lambda x.\, x)\ 1)$. Would it be possible to compute the canonical form of t?

9.5. Let us consider the following recursive definition:

$$f(x) \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } 2 \times f(x-1).$$

1. Define a well-formed, closed HOFL term t that corresponds to the above definition and determine its type.
2. Compute its denotational semantics $[\![t]\!]\rho$ and prove that

$$n \geq 0 \quad \Rightarrow \quad \textbf{let } \varphi \Leftarrow [\![t]\!]\rho.\ \varphi\lfloor n \rfloor = \lfloor 2^n \rfloor$$

Hint: Prove that the nth fixpoint approximation is

$$d_n \ = \ \lfloor\ \lambda d.\, \mathit{Cond}(\lfloor 0 \rfloor \underline{\leq}_\perp d \underline{\leq}_\perp \lfloor n \rfloor\ ,\ \lfloor 2^d \rfloor\ ,\ \perp)\ \rfloor$$

9.6. Let us consider the following recursive definition:

$$f(x) \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(f(x-1))$$

1. Define a well-formed, closed HOFL term t that corresponds to the above definition and determine its type, its canonical form and its denotational semantics.
2. Define the set of fixpoints that satisfy the recursive definition.

9.7. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec } f.\ \lambda x.\ \textbf{if } x \textbf{ then } 0 \textbf{ else } f\ (x-x)$$

1. Determine the type of t and its denotational semantics $[\![t]\!]\rho = \text{fix } \Gamma$.
2. Is fix Γ the unique fixpoint of Γ?
   Hint: Consider the elements greater than fix Γ in the order and check whether they are fixpoints for Γ.

9.8. Consider the Fibonacci sequence already found in Problem 4.14 and the corresponding term t from Problem 7.8:

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1) + F(n)$$

where $n \in \mathbb{N}$.

1. Compute a suitable transformation Γ such that $[\![t]\!]\rho = \text{fix } \Gamma$.
2. Prove that the denotational semantics $[\![t]\!]\rho$ satisfies the above equations, to conclude that the given implementation of Fibonacci numbers is correct.
   Hint: Compute $[\![(t\ 0)]\!]\rho$, $[\![(t\ 1)]\!]\rho$ and $[\![(t\ n+2)]\!]\rho$ exploiting the equality $[\![t]\!] = \Gamma\,[\![t]\!]$.

9.9. Assuming that t_1 has type τ_1, let us consider the term $t_2 \stackrel{\text{def}}{=} \lambda x.\ (t_1\ x)$.

1. Do the two terms have the same type?
2. Do the two terms have the same lazy denotational semantics?

9.10. Let us consider the terms

$$t_1 \stackrel{\text{def}}{=} \lambda x.\ \mathbf{rec}\ y.\ y+1$$
$$t_2 \stackrel{\text{def}}{=} \mathbf{rec}\ y.\ \lambda x.\ (y\ x)+2$$

1. Do the two terms have the same type?
2. Do the two terms have the same lazy denotational semantics?

9.11. Given a monotone function $f : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$, prove that $f \perp_{\mathbb{Z}_\perp} = f(f \perp_{\mathbb{Z}_\perp})$. Then, let $t : \mathit{int} \to \mathit{int}$ be a closed term of HOFL and consider the term

$$t_1 \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ (t\ (f\ x))$$

1. Determine the most general type of t_1.
2. Exploit the above result to prove that $[\![t_1]\!]\rho = [\![t_2]\!]\rho$, where

$$t_2 \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ (t\ \mathbf{rec}\ y.\ y)$$

9.12. Let us extend the syntax of (lazy) HOFL by adding the construct for sequential composition $t_1;t_2$ that, informally, represents the function obtained by applying the function t_1 to the argument and then the function t_2 to the result. Define, for the new construct

1. the typing rule;
2. the (big-step) operational semantics;
3. the denotational semantics.

Then prove that for every closed term t, the terms $(t_1;t_2\ t)$ and $(t_2\ (t_1\ t))$ have the same type and are equivalent according to the denotational semantics.

9.13. Complete the proof of the Substitution Lemma (Theorem 9.4).

9.14. Let t_1, t_2 be well-formed HOFL terms and ρ an environment.

1. Prove that

$$[\![t_1]\!]\rho \ = \ [\![t_2]\!]\rho \quad \Rightarrow \quad [\![(t_1\ x)]\!]\rho \ = \ [\![(t_2\ x)]\!]\rho \tag{9.1}$$

2. Prove that the inverse implication is generally not valid by giving a counterexample. Then, find the conditions under which the reversed implication also holds.
3. Exploit the Substitution Lemma (Theorem 9.4) to prove that for all t and $x \notin \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)$

$$[\![t_1]\!]\rho \ = \ [\![t_2]\!]\rho \quad \Rightarrow \quad [\![t[t_1/x]]\!]\rho \ = \ [\![t[t_2/x]]\!]\rho \tag{9.2}$$

4. Observe that the implication 9.1 is just a special case of the latter equality 9.2 and explain why.

9.15. Is it possible to modify the denotational semantics of HOFL assigning to the construct

$$\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1$$

- the semantics of t_1 if the semantics of t is $\perp_{N_\perp}$, and
- the semantics of t_0 otherwise? (If not, why?)

9.16. Complete the proof of Theorem 9.5.

Chapter 10
Equivalence Between HOFL Denotational and Operational Semantics

Honest disagreement is often a good sign of progress. (Mahatma Gandhi)

Abstract In this chapter we address the correspondence between the operational semantics of HOFL from Chapter 7 and its denotational semantics from Chapter 9. The situation is not as straightforward as for IMP. A first discrepancy between the two semantics is that the operational one evaluates only closed (and typable) terms, while the denotational one can handle terms with variables, thanks to environments. Apart from this minor issue, the key fact is that the canonical forms arising from the operational semantics for terms of type τ are more concrete than the mathematical elements of the corresponding domain $(V_\tau)_\perp$. Thus, it is inevitable that terms with different canonical forms can be assigned the same denotation. However, we show that terms with the same canonical form are always assigned the same denotation. Only for terms of type *int* do we have a full agreement between the two semantics. On the positive side, a term converges operationally if and only if it converges denotationally. We conclude the chapter by discussing the equivalences over terms induced by the two semantics and by presenting an alternative denotational semantics, called *unlifted*, which is simpler but less expressive than the one studied in Chapter 9.

10.1 HOFL: Operational Semantics vs Denotational Semantics

As we have done for IMP, now we address the relation between the denotational and operational semantics of HOFL. One might expect to prove a complete equivalence, as in the case of IMP:

$$\forall t, c. \quad t \to c \quad \stackrel{?}{\Leftrightarrow} \quad \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

But, as we are going to show, the situation in the case of HOFL is more complex and the implication is valid in one direction only, i.e., the operational semantics is correct but not complete:

$$t \to c \Rightarrow \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho \quad \text{ but } \quad (\forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho) \not\Rightarrow t \to c$$

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_10

Let us consider a very simple example that shows the difference between the denotational and the operational semantics.

Example 10.1. Let $c_0 = \lambda x.\ x+0$ and $c_1 = \lambda x.\ x$ be two HOFL terms, where $x : int$. Clearly

$$[\![c_0]\!]\rho = [\![c_1]\!]\rho \qquad \text{but} \qquad c_0 \not\to c_1$$

In fact, from the denotational semantics we get

$$[\![c_0]\!]\rho = [\![\lambda x.\ x+0]\!]\rho = \lfloor \lambda d.\ d \underline{+}_\perp \lfloor 0 \rfloor \rfloor = \lfloor \lambda d.\ d \rfloor = [\![\lambda x.\ x]\!]\rho = [\![c_1]\!]\rho$$

but for the operational semantics we have that both $\lambda x.\ x$ and $\lambda x.\ x+0$ are already in canonical form and $c_0 \neq c_1$.

The counterexample shows that, at least for the functional type $int \to int$, there are different canonical forms with the same denotational semantics, namely terms which compute the same function in $[\mathbb{Z}_\perp \to \mathbb{Z}_\perp]_\perp$. One might think that a refined version of our operational semantics (e.g., one which could apply an axiom like $x+0=0$) would be able to identify exactly all the canonical forms which compute the same function. However this is not possible on computability grounds: since HOFL is able to compute all computable functions, the set of canonical terms which compute the same function is not recursively enumerable, while the set of theorems of every (finite) inference system is recursively enumerable.

Even if we cannot have a strong correspondence result between the operational and denotational semantics as was the case for IMP, we can establish a full agreement between the two semantics w.r.t. the notion of termination. In particular, by letting the predicate $t\downarrow$ denote the fact that the term t can be reduced to some canonical form (called *operational convergence*) and $t \Downarrow$ denote the fact that the term $t : \tau$ is assigned a denotation other than $\perp_{(V_\tau)_\perp}$ (called *denotational convergence*), we have the perfect match:

$$t\downarrow \quad \Leftrightarrow \quad t \Downarrow$$

10.2 Correctness

We are ready to show the correctness of the operational semantics of HOFL w.r.t. the denotational one. Note that since the operational semantics is defined for closed terms only, the environment is inessential in the following theorem.

Theorem 10.1 (Correctness). *Let $t : \tau$ be a HOFL closed term and let $c : \tau$ be a canonical form. Then we have*

$$t \to c \quad \Rightarrow \quad \forall \rho \in Env.\ [\![t]\!]\rho = [\![c]\!]\rho$$

Proof. We proceed by rule induction. So we prove

$$P(t \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

for the conclusion $t \to c$ of each rule, when the predicate holds for the premises.

C_τ: The rule for terms in canonical forms (integers, pairs, abstraction) is

$$\frac{}{c \to c}$$

We have to prove $P(c \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![c]\!]\rho = [\![c]\!]\rho$, which is obviously true.

Arit.: Let us consider the rules for arithmetic operators $\text{op} \in \{+, -, \times\}$:

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1 \mathrel{\underline{\text{op}}} n_2}$$

We assume the inductive hypotheses:

$$P(t_1 \to n_1) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1]\!]\rho = [\![n_1]\!]\rho = \lfloor n_1 \rfloor$$
$$P(t_2 \to n_2) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_2]\!]\rho = [\![n_2]\!]\rho = \lfloor n_2 \rfloor$$

and we want to prove

$$P(t_1 \text{ op } t_2 \to n_1 \mathrel{\underline{\text{op}}} n_2) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1 \text{ op } t_2]\!]\rho = \left[\!\!\left[n_1 \mathrel{\underline{\text{op}}} n_2 \right]\!\!\right] \rho$$

We have

$$\begin{aligned}
[\![t_1 \text{ op } t_2]\!]\rho &= [\![t_1]\!]\rho \mathrel{\underline{\text{op}}_\perp} [\![t_2]\!]\rho && \text{(by definition of } [\![\cdot]\!]) \\
&= \lfloor n_1 \rfloor \mathrel{\underline{\text{op}}_\perp} \lfloor n_2 \rfloor && \text{(by inductive hypotheses)} \\
&= \lfloor n_1 \mathrel{\underline{\text{op}}} n_2 \rfloor && \text{(by definition of } \underline{\text{op}}_\perp) \\
&= \left[\!\!\left[n_1 \mathrel{\underline{\text{op}}} n_2 \right]\!\!\right] \rho && \text{(by definition of } [\![\cdot]\!])
\end{aligned}$$

Cond.: In the case of the conditional construct we have two rules to consider. For

$$\frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0}$$

we can assume

$$P(t \to 0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![0]\!]\rho = \lfloor 0 \rfloor$$
$$P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$$

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

$$\begin{aligned}
[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho &= \mathit{Cond}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by def. of } [\![\cdot]\!]) \\
&= \mathit{Cond}(\lfloor 0 \rfloor, [\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by ind. hyp.)} \\
&= [\![t_0]\!]\rho && \text{(by def. of } \mathit{Cond}) \\
&= [\![c_0]\!]\rho && \text{(by ind. hyp.)}
\end{aligned}$$

An analogous argument holds for the second rule of the conditional operator.

Proj.: Let us consider the rule for the first projection:

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0}$$

We can assume

$$P(t \to (t_0, t_1)) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![(t_0, t_1)]\!]\rho$$
$$P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$$

and we want to prove

$$P(\textbf{fst}(t) \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![\textbf{fst}(t)]\!]\rho = [\![c_0]\!]\rho$$

We have

$$\begin{aligned}
[\![\textbf{fst}(t)]\!]\rho &= \pi_1^*([\![t]\!]\rho) && \text{(by def. of } [\![\cdot]\!]) \\
&= \pi_1^*([\![(t_0, t_1)]\!]\rho) && \text{(by ind. hyp.)} \\
&= \pi_1^*(\lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor) && \text{(by def. of } [\![\cdot]\!]) \\
&= \pi_1([\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by def. of lifting)} \\
&= [\![t_0]\!]\rho && \text{(by def. of } \pi_1) \\
&= [\![c_0]\!]\rho && \text{(by ind. hyp.)}
\end{aligned}$$

An analogous argument holds for the **snd** operator.

App.: The rule for application is

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

We can assume

$$P(t_1 \to \lambda x.\, t_1') \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1]\!]\rho = [\![\lambda x.\, t_1']\!]\rho$$
$$P(t_1'[{}^{t_0}/_x] \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![c]\!]\rho$$

and we want to prove

$$P((t_1\ t_0) \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![(t_1\ t_0)]\!]\rho = [\![c]\!]\rho$$

We have

$$\begin{aligned}
[\![(t_1\ t_0)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho) && \text{(by definition of } [\![\cdot]\!]) \\
&= \textbf{let}\ \varphi \Leftarrow [\![\lambda x.\ t_1']\!]\rho.\ \varphi([\![t_0]\!]\rho) && \text{(by ind. hypothesis)} \\
&= \textbf{let}\ \varphi \Leftarrow \lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \rfloor.\ \varphi([\![t_0]\!]\rho) && \text{(by definition of } [\![\cdot]\!]) \\
&= (\lambda d.\ [\![t_1']\!]\rho[^d/_x])\ ([\![t_0]\!]\rho) && \text{(by de-lifting)} \\
&= [\![t_1']\!]\rho[^{[\![t_0]\!]\rho}/_x] && \text{(by application)} \\
&= [\![t_1'[^{t_0}/_x]]\!]\rho && \text{(by Subst. Lemma)} \\
&= [\![c]\!]\rho && \text{(by ind. hypothesis)}
\end{aligned}$$

Rec.: Finally, we consider the rule for recursion:

$$\frac{t[^{\textbf{rec}\ x.\ t}/_x] \to c}{\textbf{rec}\ x.\ t \to c}$$

We can assume

$$P(t[^{\textbf{rec}\ x.\ t}/_x] \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![t[^{\textbf{rec}\ x.\ t}/_x]]\!]\rho = [\![c]\!]\rho$$

and we want to prove

$$P(\textbf{rec}\ x.\ t \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![\textbf{rec}\ x.\ t]\!]\rho = [\![c]\!]\rho.$$

We have

$$\begin{aligned}
[\![\textbf{rec}\ x.\ t]\!]\rho &= [\![t]\!]\rho[^{[\![\textbf{rec}\ x.\ t]\!]\rho}/_x] && \text{(by definition)} \\
&= [\![t[^{\textbf{rec}\ x.\ t}/_x]]\!]\rho && \text{(by the Substitution Lemma)} \\
&= [\![c]\!]\rho && \text{(by inductive hypothesis)}
\end{aligned}$$

Since there are no more rules to consider, we conclude the thesis holds. □

10.3 Agreement on Convergence

Now we define the concept of convergence (i.e., termination) for the operational and the denotational semantics.

Definition 10.1 (Operational convergence). Let $t : \tau$ be a closed term of HOFL. We define the following predicate:

$$t\downarrow \quad \Leftrightarrow \quad \exists c \in C_\tau.\ t \longrightarrow c$$

If $t\downarrow$, then we say that t *converges operationally*. We say that t *diverges*, written $t\uparrow$, if t does not converge operationally.

A term t converges operationally if the term can be evaluated to a canonical form c. For the denotational semantics we have that a term t converges if the evaluation function applied to t takes a value different from $\bot$.

Definition 10.2 (Denotational convergence). Let t be a closed term of HOFL with type τ. We define the following predicate:

$$t \Downarrow \quad \Leftrightarrow \quad \forall \rho \in Env, \exists v \in V_\tau.\ [\![t]\!]\rho = \lfloor v \rfloor$$

If the predicate holds for t then we say that t *converges denotationally*.

We aim to prove that the two semantics agree at least on the notion of convergence. The implication $t \downarrow \Rightarrow\ t \Downarrow$ can be readily proved.

Theorem 10.2. *Let $t : \tau$ be a closed typable term of HOFL. Then we have*

$$t \downarrow \quad \Rightarrow \quad t \Downarrow$$

Proof. If $t \to c$, then $\forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$ by Theorem 10.1. But $[\![c]\!]\rho$ is a lifted value, (see Theorem 9.6) and thus it is different from $\bot_{(V_\tau)_\bot}$. □

Also the opposite implication $t \Downarrow \Rightarrow\ t \downarrow$ holds (for any closed and typable term t, see Theorem 10.3) but the proof is not straightforward: we cannot simply rely on structural induction; instead it is necessary to introduce a particular logical relation between elements of the interpretation domains and HOFL terms. We will only sketch the proof, but first we show that the standard structural induction does not help in proving the agreement of semantics about convergence.

Remark 10.1 (On the reason why structural induction fails for proving $t \Downarrow \Rightarrow\ t \downarrow$). The property $P(t) \stackrel{\text{def}}{=} t \Downarrow \Rightarrow\ t \downarrow$ cannot be proved by structural induction on t. Here we give some insights on the reason why this is so. Let us focus on the case of function application $(t_1\ t_0)$. By structural induction, we assume

$$P(t_1) \stackrel{\text{def}}{=} t_1 \Downarrow \Rightarrow\ t_1 \downarrow \qquad \text{and} \qquad P(t_0) \stackrel{\text{def}}{=} t_0 \Downarrow \Rightarrow\ t_0 \downarrow$$

and we want to prove $P(t_1\ t_0) \stackrel{\text{def}}{=} (t_1\ t_0) \Downarrow \Rightarrow\ (t_1\ t_0) \downarrow$.

Let us assume the premise $(t_1\ t_0) \Downarrow$ (i.e., $[\![(t_1\ t_0)]\!]\rho \neq \bot$) of the implication. We would like to prove that $(t_1\ t_0) \downarrow$, i.e., that $\exists c.\ (t_1\ t_0) \to c$. By the definition of the denotational semantics we have $t_1 \Downarrow$. In fact

$$[\![(t_1\ t_0)]\!]\rho \stackrel{\text{def}}{=} \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho)$$

and therefore $[\![(t_1\ t_0)]\!]\rho \neq \bot$ requires $[\![t_1]\!]\rho \neq \bot$. By the first inductive hypothesis we then have $t_1 \downarrow$ and by the definition of the operational semantics it must be the case that $t_1 \to \lambda x.\ t_1'$ for some x and t_1'. By correctness (Theorem 10.1), we then have

$$[\![t_1]\!]\rho = [\![\lambda x.\ t_1']\!]\rho = \left\lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \right\rfloor$$

Therefore

$$\begin{aligned}
[\![(t_1\ t_0)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow \lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \rfloor .\ \varphi([\![t_0]\!]\rho) && \text{(see above)}\\
&= (\lambda d.\ [\![t_1']\!]\rho[^d/_x])\ ([\![t_0]\!]\rho) && \text{(by de-lifting)}\\
&= [\![t_1']\!]\rho[^{[\![t_0]\!]\rho}/_x] && \text{(by functional application)}\\
&= [\![t_1'[^{t_0}/_x]]\!]\rho && \text{(by the Substitution Lemma)}
\end{aligned}$$

So $(t_1\ t_0)\Downarrow$ if and only if $t_1'[^{t_0}/_x]\Downarrow$. We would like to conclude by structural induction that $t_1'[^{t_0}/_x]\downarrow$ and then prove the theorem by using the rule

$$\frac{t_1 \to \lambda x.\ t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

but this is incorrect since $t_1'[^{t_0}/_x]$ is not a subterm of $(t_1\ t_0)$ and we are not allowed to assume that $P(t_1'[^{t_0}/_x])$ holds.

Theorem 10.3. *For any closed typable term $t : \tau$ we have:*

$$t\Downarrow \quad\Rightarrow\quad t\downarrow$$

Proof. The proof exploits two suitable *logical relations*, indexed by HOFL types:

- $\lesssim^c_\tau \subseteq V_\tau \times C_\tau$, which relates canonical forms to corresponding values in V_τ and is defined by structural induction over types τ;
- $\lesssim_\tau \subseteq (V_\tau)_\perp \times T_\tau$, which relates well-formed (closed) terms to values in $(V_\tau)_\perp$ and is defined by letting

$$d \lesssim_\tau t \quad \stackrel{\text{def}}{=} \quad \forall v \in V_\tau.\ d = \lfloor v \rfloor \Rightarrow \exists c.\ t \to c \wedge v \lesssim^c_\tau c$$

  In particular, note that, by definition, we have $\bot_{(V_\tau)_\perp} \lesssim_\tau t$ for any term $t : \tau$.

The logical relation on canonical forms is defined as follows:

ground type:	we simply let $n \lesssim^c_{int} n$;
product type:	we let $(d_0,d_1) \lesssim^c_{\tau_0*\tau_1} (t_0,t_1)$ iff $d_0 \lesssim_{\tau_0} t_0$ and $d_1 \lesssim_{\tau_1} t_1$;
function type:	we let $\varphi \lesssim^c_{\tau_0\to\tau_1} \lambda x.\ t$ iff $\forall d_0 \in (V_{\tau_0})_\perp$ and $\forall t_0 : \tau_0$ closed, $d_0 \lesssim_{\tau_0} t_0$ implies $\varphi(d_0) \lesssim_{\tau_1} t[^{t_0}/_x]$.

Then one can show by structural induction on $t : \tau$ that

1. $\forall d,d' \in (V_\tau)_\perp.\ (d \sqsubseteq_{(V_\tau)_\perp} d' \wedge d' \lesssim_\tau t) \Rightarrow d \lesssim_\tau t$;
2. if $\{d_i\}_{i\in\mathbb{N}}$ is a chain in $(V_\tau)_\perp$ such that $\forall i \in \mathbb{N}.\ d_i \lesssim_\tau t$, we have $\bigsqcup_{i\in\mathbb{N}} d_i \lesssim_\tau t$ (i.e., the predicate $\cdot \lesssim_\tau t$ is inclusive).

Finally, by structural induction on terms, one can prove that $\forall t : \tau$ with $\mathrm{fv}(t) \subseteq \{x_1 : \tau_1, \ldots, x_k : \tau_k\}$, if $\forall i \in [1,k].\ d_i \lesssim_{\tau_i} t_i$ then $[\![t]\!]\rho[^{d_1}/_{x_1},\ldots,^{d_k}/_{x_k}] \lesssim_\tau t[^{t_1}/_{x_1},\ldots,^{t_k}/_{x_k}]$. In fact, taking $t : \tau$ closed, it follows from the definition of $\lesssim_\tau$ that if $t\Downarrow$, i.e., $[\![t]\!]\rho = \lfloor v \rfloor$ for some $v \in V_\tau$, then $t \to c$ for some canonical form c, i.e., $t\downarrow$. □

10.4 Operational and Denotational Equivalences of Terms

In Section 10.1 we have shown that the denotational semantics is more abstract than the operational. In order to study the relationship between the operational and denotational semantics of HOFL we now introduce two equivalence relations between terms. Operationally two closed terms are equivalent if they both diverge or have the same canonical form.

Definition 10.3 (Operational equivalence). Let t_0 and t_1 be two well-typed terms of HOFL. We define a binary relation $\equiv_{op}$ as follows:

$$t_0 \equiv_{op} t_1 \quad \Leftrightarrow \quad (t_0 \uparrow \wedge t_1 \uparrow) \vee (\exists c.\, t_0 \to c \wedge t_1 \to c)$$

We say that t_0 is *operationally equivalent* to t_1 if $t_0 \equiv_{op} t_1$.

We have also the denotational counterpart of the definition of equivalence.

Definition 10.4 (Denotational equivalence). Let t_0 and t_1 be two well-typed terms of HOFL. We define a binary relation $\equiv_{den}$ as follows:

$$t_0 \equiv_{den} t_1 \quad \Leftrightarrow \quad \forall \rho.\, [\![t_0]\!]\rho = [\![t_1]\!]\rho$$

We say that t_0 is *denotationally equivalent* to t_1 if $t_0 \equiv_{den} t_1$.

Remark 10.2. Note that the definition of denotational equivalence applies also to non-closed terms. Operational equivalence of non-closed terms t and t' could also be defined by taking the closure of the equivalence w.r.t. the embedding of t and t' in any context $C[\cdot]$ such that $C[t]$ and $C[t']$ are also closed, i.e., by requiring that $C[t]$ and $C[t']$ are operationally equivalent for any context $C[\cdot]$.

From Theorem 10.1 it follows that $\equiv_{op} \Rightarrow \equiv_{den}$.

As pointed out in Example 10.1, we know that $\equiv_{den} \not\Rightarrow \equiv_{op}$.

So it is in this sense that we can say that the denotational semantics is *more abstract* then the operational one, because the former identifies more terms than the latter. Note that if we assume $t_0 \equiv_{den} t_1$ and $[\![t_0]\!]\rho \neq \bot$ then we can only conclude that $t_0 \to c_0$ and $t_1 \to c_1$ for some canonical forms c_0 and c_1. We have $[\![c_0]\!]\rho = [\![c_1]\!]\rho$, but nothing ensures that $c_0 = c_1$ (see Example 10.1 at the beginning of this chapter).

Only when we restrict our attention to the terms of HOFL that are typed as integers do the corresponding operational and denotational semantics fully agree. This is because if c_0 and c_1 are canonical forms in C_{int} then it holds that $[\![c_0]\!]\rho = [\![c_1]\!]\rho \Leftrightarrow c_0 = c_1$. It can be proved that *int* is the only type for which the full correspondence holds.

Theorem 10.4. *Let t : int be a closed term of HOFL and $n \in \mathbb{Z}$. Then*

$$\forall \rho.\, [\![t]\!]\rho = \lfloor n \rfloor \quad \Leftrightarrow \quad t \to n$$

Proof. We prove the two implications separately.

$\Rightarrow$) If $[\![t]\!]\rho = \lfloor n \rfloor$, then $t \Downarrow$ and thus $t \downarrow$ by the soundness of the denotational semantics, namely $\exists m$ such that $t \to m$, but then $[\![t]\!]\rho = \lfloor m \rfloor$ by Theorem 10.1, thus $n = m$ and $t \to n$.

$\Leftarrow$) Just Theorem 10.1, because $[\![n]\!]\rho = \lfloor n \rfloor$. □

10.5 A Simpler Denotational Semantics

We conclude this chapter by presenting a simpler denotational semantics which we call *unlifted*, because it does not use the lifted domains. This semantics is simpler but also less expressive than the lifted one. We define the following new domains:

$$D_{int} \stackrel{\text{def}}{=} \mathbb{Z}_\perp \qquad D_{\tau_1 * \tau_2} \stackrel{\text{def}}{=} D_{\tau_1} \times D_{\tau_2} \qquad D_{\tau_1 \to \tau_2} \stackrel{\text{def}}{=} [D_{\tau_1} \to D_{\tau_2}]$$

Now we can let $Env' \stackrel{\text{def}}{=} Var \to \bigcup_\tau D_\tau$ and define the simpler interpretation function $[\![t : \tau]\!]' : Env' \to D_\tau$ as follows (where $\rho \in Env'$):

(exactly as before)

$$\begin{aligned}
[\![n]\!]'\rho &= \lfloor n \rfloor \\
[\![x]\!]'\rho &= \rho(x) \\
[\![t_1 \text{ op } t_2]\!]'\rho &= [\![t_1]\!]'\rho \ \underline{\text{op}}_\perp \ [\![t_2]\!]'\rho \\
[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]'\rho &= \mathit{Cond}([\![t_0]\!]'\rho, [\![t_1]\!]'\rho, [\![t_2]\!]'\rho) \\
[\![\textbf{rec } x.\, t]\!]'\rho &= \text{fix}\, \lambda d.\ [\![t]\!]'\rho[^d/_x]
\end{aligned}$$

(updated definitions)

$$\begin{aligned}
[\![(t_1, t_2)]\!]'\rho &= ([\![t_1]\!]'\rho, [\![t_2]\!]'\rho) \\
[\![\textbf{fst}(t)]\!]'\rho &= \pi_1([\![t]\!]'\rho) \\
[\![\textbf{snd}(t)]\!]'\rho &= \pi_2([\![t]\!]'\rho) \\
[\![\lambda x.\, t]\!]'\rho &= \lambda d.\ [\![t]\!]'\rho[^d/_x] \\
[\![(t_1\ t_2)]\!]'\rho &= ([\![t_1]\!]'\rho)\ ([\![t_2]\!]'\rho)
\end{aligned}$$

Note that the "unlifted" semantics differ from the "lifted" one only in the cases of pairing, projections, abstraction and application. On the one hand the unlifted denotational semantics is much simpler to read than the lifted one. On the other hand the unlifted semantics is more abstract than the lifted one and cannot express some interesting properties. For instance, consider the two HOFL terms

$$t_1 \stackrel{\text{def}}{=} \textbf{rec } x.\, x : \mathit{int} \to \mathit{int} \quad \text{and} \quad t_2 \stackrel{\text{def}}{=} \lambda x.\ \textbf{rec } y.\, y : \mathit{int} \to \mathit{int}$$

In the lifted semantics we have $[\![t_1]\!]\rho = \perp_{[\mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp]_\perp}$ and $[\![t_2]\!]\rho = \lfloor \perp_{[\mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp]} \rfloor$, thus

$$t_1 \not\Downarrow \quad \text{and} \quad t_2 \Downarrow$$

In the unlifted semantics $[\![t_1]\!]'\rho = [\![t_2]\!]'\rho = \perp_{[\mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp]}$, thus

$$t_1 \not\Downarrow' \quad \text{and} \quad t_2 \not\Downarrow'$$

Note however that $t_1 \uparrow$ while $t_2 \downarrow$, thus the property $t \downarrow \Rightarrow t \Downarrow'$ does not hold, at least for some $t : int \to int$, since $t_2 \downarrow$ but $t_2 \not\Downarrow'$. However, the property holds for the unlifted semantics in the case of integers.

As a concluding remark, we observe that the existence of two different denotational semantics for HOFL, both reasonable, shows that denotational semantics is, to some extent, an arbitrary construction, which depends on the properties one wants to express.

Problems

10.1. Prove that the HOFL terms

$$t_1 \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ ((\lambda y.\ 1)\ (f\ x)) \qquad t_2 \stackrel{\text{def}}{=} \lambda x.\ 1$$

have the same type and the same denotational semantics but different canonical forms.

10.2. Let us consider the HOFL term

$$map \stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ ((f\ \mathbf{fst}(x)), (f\ \mathbf{snd}(x)))$$

from Problem 7.5.

1. Give the denotational semantics of *map* and of $(map\ \lambda z.\ z)$.
2. Give two terms $t_1 : int$ and $t_2 : int$ such that the terms

$$((map\ \lambda z.\ z)(t_1, t_2)) \qquad ((map\ \lambda z.\ z)(t_2, t_1))$$

have different canonical forms but the same denotational semantics.

10.3. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ ((\lambda y.\ \mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0)\ x)$$

from Problem 7.6. Compute its denotational semantics, checking the equivalence with its operational semantics.

10.4. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \mathbf{if}\ \mathbf{fst}(x) \times \mathbf{snd}(x)\ \mathbf{then}\ x\ \mathbf{else}\ (f\ (f\ x))$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.5. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{fst}(x) - \textbf{snd}(x)\ \textbf{then}\ x\ \textbf{else}\ (f\ x)$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.6. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ F.\ \lambda f.\ \lambda n.\ \textbf{if}\ (f\ n)\ \textbf{then}\ 0\ \textbf{else}\ ((F\ f)\ n)$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.7. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ (\textbf{fst}(x)\ \textbf{snd}(x))\ \textbf{then}\ x\ \textbf{else}\ (f\ x)$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.8. Modify the ordinary HOFL semantics by defining the denotational semantics of the conditional construct as follows

$$[\![\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1]\!]\rho = \mathit{Condd}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

where

$$\mathit{Condd}(z, z_0, z_1) = \begin{cases} z_0 & \text{if } z = \lfloor 0 \rfloor \ \vee\ z_0 = z_1 \\ z_1 & \text{if } z = \lfloor n \rfloor \ \wedge\ n \neq 0 \\ \bot & \text{otherwise} \end{cases}$$

Assume that $t_0, t_1 : \mathit{int}$.

1. Prove that *Condd* is a monotonic, continuous function.
2. Show a HOFL term with a different semantics than the ordinary, and explain how the relation between operational and denotational semantics of HOFL is actually changed.

10.9. Modify the semantics of HOFL assuming the following operational semantics for the conditional command:

$$\frac{t_0 \to 0 \quad t_1 \to c_1 \quad t_2 \to c_2}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \to\ c_1} \qquad \frac{t_0 \to n \quad n \neq 0 \quad t_1 \to c_1 \quad t_2 \to c_2}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \to\ c_2}.$$

1. Exibit the corresponding denotational semantics.
2. Prove that also for the modified semantics it holds that $t \to c$ implies $[\![t]\!] = [\![c]\!]$.

3. Finally, compute the operational and the denotational semantics of $(\mathit{fact}\ 0)$, with

$$\mathit{fact} \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\ (x-1))$$

and check whether they coincide.

10.10. Suppose the operational semantics of projections is changed

$$\text{from} \quad \frac{t \to (t_1, t_2) \quad t_1 \to c}{\mathbf{fst}(t) \to c} \qquad \text{to} \quad \frac{t \to (t_1, t_2) \quad t_1 \to c \quad t_2 \to c'}{\mathbf{fst}(t) \to c}$$

and analogously for **snd**, without changing the denotational semantics.

1. Prove that the property $t \to c \Rightarrow [\![t]\!]\rho = [\![c]\!]\rho$ is still valid.
2. Exhibit a counterexample showing that the property $[\![t]\!]\rho \neq \bot \Rightarrow t \to c$ is no longer valid.
3. Finally, modify the denotational semantics to recover the above property and illustrate its validity for the counterexample previously proposed.

10.11. Modify the operational semantics of HOFL by taking the following rules for conditionals:

$$\frac{t \to 0 \quad t_0 \to c_0 \quad t_1 \to c_1}{\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_0 \to c_0 \quad t_1 \to c_1}{\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1 \to c_1}$$

without changing the denotational semantics. Prove that

1. for any term t and canonical form c, we have $t \to c\ \Rightarrow\ \forall\rho.\ [\![t]\!]\rho = [\![c]\!]\rho$;
2. in general $t \Downarrow \not\Rightarrow t \downarrow$ (and exhibit a counterexample).

10.12. Suppose we extend HOFL with the inference rule

$$\frac{t_1 \to 0}{t_1 \times t_2 \to 0}$$

as in Problem 7.12.

1. Exhibit a counterexample showing that the property

$$\forall t, c. \quad t \to c \quad \Rightarrow \quad \forall\rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

is no longer valid.
2. Modify the denotational semantics so that the above correspondence is obtained, and prove that this is the case.
3. Repeat the exercise adding also the inference rule

$$\frac{t_2 \to 0}{t_1 \times t_2 \to 0}$$

10.13. Prove formally that

$$\text{if} \quad x \notin \mathrm{fv}(t) \quad \text{then} \quad \mathbf{rec}\ x.\ t \text{ is equivalent to } t$$

employing both the operational and the denotational semantics.

10.14. Assume that the HOFL term t_0 has c_0 as canonical form.

1. Exploit the Substitution Lemma (Theorem 9.4) to prove that for every term t_1' we have
$$[\![t_1'[{}^{t_0}/_x]]\!]\,\rho \;=\; [\![t_1'[{}^{c_0}/_x]]\!]\,\rho$$
2. Prove that if $t_1' : \mathit{int}$ and $\mathrm{fv}(t_1') \subseteq \{x\}$, then $t_1'[{}^{t_0}/_x] \equiv_{op} t_1'[{}^{c_0}/_x]$.
3. Conclude that if we replace the lazy evaluation rule
$$\frac{t_1 \to \lambda x.\ t_1' \quad t_1'[t_0/x] \to c}{(t_1\ t_0) \to c}$$
with the eager rule
$$\frac{t_1 \to \lambda x.\ t_1' \quad t_0 \to c_0 \quad t_1'[c_0/x] \to c}{(t_1\ t_0) \to c}$$
then, if $(t_1\ t_0) \to c : \mathit{int}$ in the eager semantics, then $(t_1\ t_0) \to c$ in the lazy semantics.
4. Exhibit a simple counterexample such that $\exists c.\ (t_1\ t_0) \to c$ according to the lazy semantics but not to the eager one.
5. Finally, exhibit another counterexample where the type of t_1' is not *int* and the properties in points 2 and 3 do not hold.

10.15. Extend the operational semantics of HOFL to non-closed terms, by allowing canonical forms that are not closed but otherwise keeping the same inference rules. Show an example of reduction to canonical form for a non-closed term t. Then, prove that the following properties are still valid:

1. subject reduction: $t : \tau$ and $t \to c$ implies $c : \tau$;
2. $t \to c$ implies $[\![t]\!]\,\rho = [\![c]\!]\,\rho$ (recall that the Substitution Lemma holds for any terms, including non-closed ones);
3. $t \downarrow$ implies $t \Downarrow$;
4. $t_1 : \mathit{int} \to c_1$, $t_2 : \mathit{int} \to c_2$ and $[\![t_1]\!] = [\![t_2]\!]$ imply $c_1 = c_2$;
5. $t \to c$ implies $[\![t[{}^{\mathbf{rec}\ z.\ z}/_x]]\!]\,\rho = [\![c[{}^{\mathbf{rec}\ z.\ z}/_x]]\!]\,\rho$.
Hint: Exploit property 2 above and the Substitution Lemma.

10.16. Modify the denotational semantics of HOFL by restricting the use of the *lifting* domain construction only to integers, namely $V_{\mathit{int}} = \mathbb{Z}_\perp$ but $V_{\tau_1 * \tau_2} = V_{\tau_1} \times V_{\tau_2}$ and similarly for functions.

1. List all the modified clauses of the denotational semantics.
2. Prove that $t \to c$ implies $[\![t]\!]\,\rho = [\![c]\!]\,\rho$.
3. Finally, prove that it is not true that $t \to c$ implies $[\![t]\!]\,\rho \neq \perp$.
Hint: consider the HOFL term $t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ (f\ x) \ :\ \mathit{int} \to \mathit{int}$.

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, the 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 of 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 nondeterminism and non-termination are desirable semantic 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, the necessity emerges to deal with

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_11

Nondeterminism:	Nondeterminism is needed to model time races between different signals and to abstract away from programming details which are irrelevant for the interactive behaviour of systems.
Parallelism:	Parallelism allows agents to perform tasks independently. For our purposes, this will be modelled by using nondeterministic 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 us 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 nondeterministic 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 nondeterministic systems with non-trivial infinite behaviours are not inclusive (as is the case with 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** or the HOFL term **rec** $x.\ x$) is assigned the denotation $\bot$, hence all diverging programs are considered equivalent. Labelled transition systems allow us 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 us to express and prove the relevant properties of the modelled system.

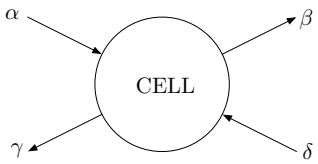
Let us show how CCS works with an example.

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 destroyed 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 such process 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 parameterised by 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. Then the agent checks whether the received value 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 nondeterministically 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 nondeterministic 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 the old stored value v to the right neighbour 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 nondeterministic 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, which 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 neighbouring 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 place the value 1 on the stack. This can be achieved by sending the value 1 on the channel α to the cell ENDCELL (see Figure 11.1).

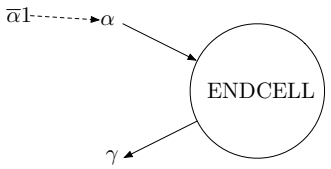


Fig. 11.1: ENDCELL$\backslash\beta\backslash\delta$ receiving the value 1 on channel α

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

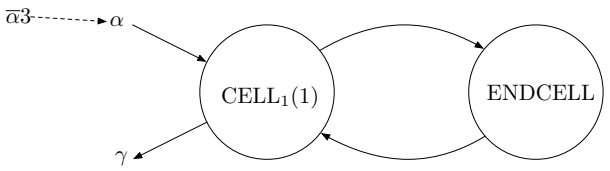


Fig. 11.2: $(\mathrm{CELL}_1(1) \frown \mathrm{ENDCELL})\backslash\beta\backslash\delta$ receiving the value 3 on channel α

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

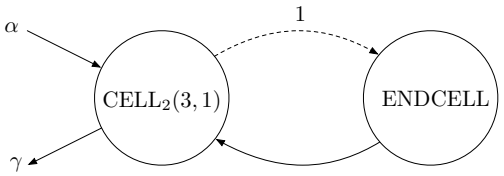


Fig. 11.3: $(\mathrm{CELL}_2(3,1) \frown \mathrm{ENDCELL})\backslash\beta\backslash\delta$ before right-shifting the value 1

Then, when the rightmost cell (ENDCELL) receives the value 1 on its channel α, privately connected to the channel β of the leftmost cell ($\mathrm{CELL}_2(3,1)$) via the linking operation $\frown$, it will change its state to $\mathrm{CELL}_1(1)$ 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.

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

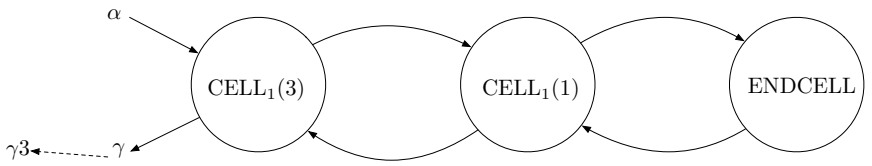


Fig. 11.4: $\mathrm{CELL}_1(3) \supset \mathrm{CELL}_1(1) \supset \mathrm{ENDCELL} \backslash \beta \backslash \delta$ before a pop operation

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

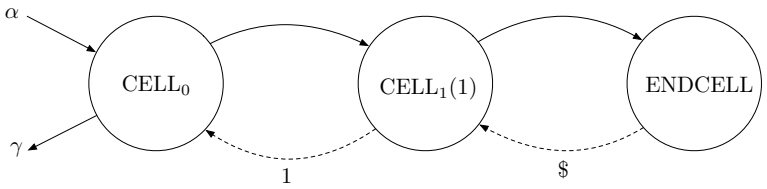


Fig. 11.5: $(\mathrm{CELL}_0 \supset \mathrm{CELL}_1(1) \supset \mathrm{ENDCELL}) \backslash \beta \backslash \delta$ before left-shifting value 1

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).

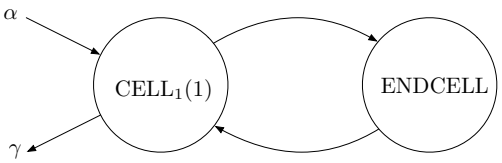


Fig. 11.6: $(\mathrm{CELL}_1(1) \supset \mathrm{ENDCELL} \supset \mathbf{nil}) \backslash \beta \backslash \delta$

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

CCS was introduced by Turing Award winner Robin Milner (1934–2010) in the early 1980s. 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 cannot be used for synchronisation. Moreover we will use the following conventions:

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

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

Now we are ready to present the syntax of CCS.

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

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

We briefly comment on 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 nondeterministically to behave like 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 recall 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 nondeterministic 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 nondeterminism appears. Unlike the case of sum, where nondeterminism is a characteristic of the modelled system, here nondeterminism is a characteristic of the semantic style that allows p and q to interleave their actions in $p \mid q$, i.e., nondeterminism is exploited to model the parallel behaviour of the system:

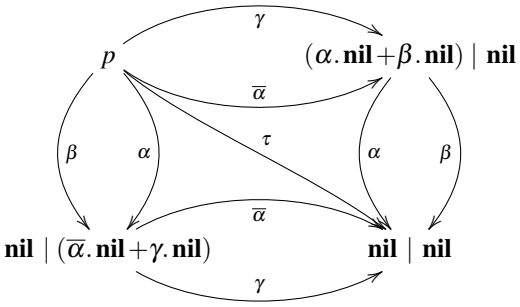
$$\text{(Par)} \qquad \frac{p \xrightarrow{\mu} p'}{p \mid q \xrightarrow{\mu} p' \mid q} \qquad \frac{q \xrightarrow{\mu} q'}{p \mid q \xrightarrow{\mu} p \mid q'}$$

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

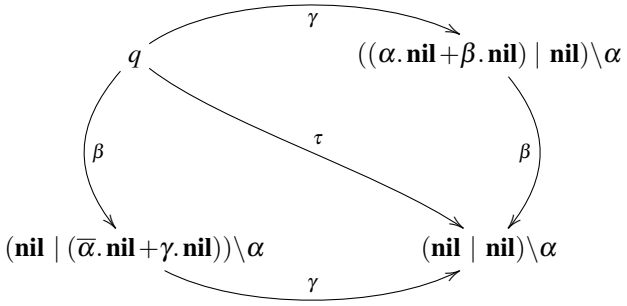
$$\text{(Com)} \qquad \frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\overline{\lambda}} q_2}{p_1 \mid q_1 \xrightarrow{\tau} p_2 \mid q_2}$$

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

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

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 that involves α- and β-transitions only. 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.

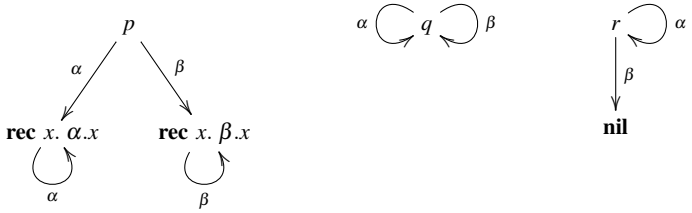


Fig. 11.7: The LTSs of three recursively defined processes

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

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

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.

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 $p_1,\ldots,p_n$ and process names $x_1,\ldots,x_n$

$$G(p,X) \wedge \bigwedge_{i\in[1,n]} G(p_i,X) \quad \Rightarrow \quad G(p[{}^{p_1}/_{x_1},\cdots,{}^{p_n}/_{x_n}],X)$$

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

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

where we have annotated each derivation step with the name of the applied rule. Thus, $r \xrightarrow{\overline{\alpha}} r$ and since no other rule is 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, 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 (nondeterministic) 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.,

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

from which we derive

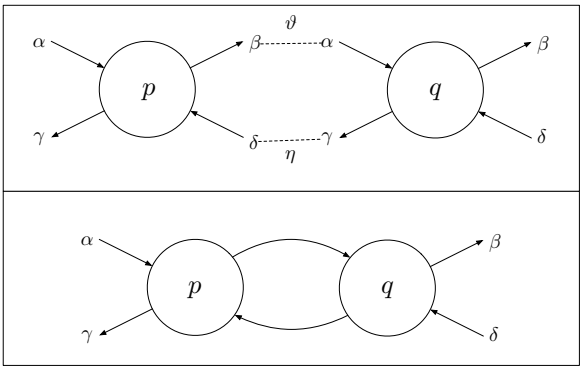


Fig. 11.8: Graphically illustration of the concatenation operator $p \frown q$

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

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 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, which views ϑ as α. Instead, messages sent on β by q are not redirected to ϑ and will appear as messages sent on β by the whole process $p \frown q$ (see Figure 11.8).

11.3.8 CCS with Value Passing

Example 11.1 considers I/O operations where values can be received and transmitted. This would correspond to extending 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 $\alpha(x).p$, the symbol x is bound with scope p. Assuming a set of possible values V is 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 there are 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 such as

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

can be represented by the CCS process

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

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

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

11.3.9 Recursive Declarations and the Recursion Operator

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

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

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

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

so that in r_i all occurrences of X_j occur under a recursion operator $\mathbf{rec}\ X_j$ 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

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

From which we derive

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

11.4 Abstract Semantics of CCS

In the previous section we have defined a mapping from CCS agents to LTSs, i.e., to a special class of labelled graphs. It is easy to see that such an 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 they exhibit the same behaviour. Analogously for $p+q$ and $q+p$ or for $p+\mathbf{nil}$ and p. Thus it is important to find a good notion of equivalence, able to provide a more abstract semantics for CCS. As 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.

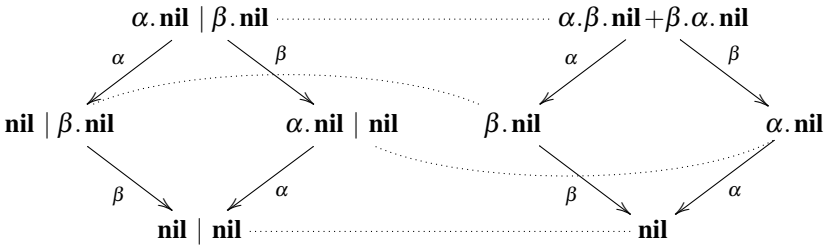
In this section, we first show that neither graph isomorphism nor trace equivalence offers fully satisfactory abstract semantics for CCS. Next, we introduce a more appropriate abstract semantics of CCS by defining a relation, called *strong bisimilarity*, that captures the ability of processes to simulate 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 defining 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 $\alpha.\mathbf{nil} \mid \beta.\mathbf{nil}$ and $\alpha.\beta.\mathbf{nil}+\beta.\alpha.\mathbf{nil}$. 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 equivalent. This result is surprising, since they have rather different structures. In fact, the example shows that concurrency can be reduced to nondeterminism 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:

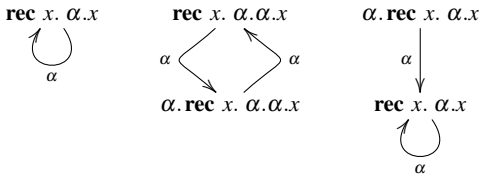


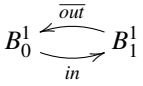
Fig. 11.9: Three non-isomorphic agents

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 an infinite sequence of α-transitions.

Example 11.8 (Buffers). Let us denote by B_k^n a buffer of capacity n of which k positions are busy. For example, to represent a buffer of capacity 1 in CCS one can let (using recursive definitions)

$$B_0^1 \stackrel{\text{def}}{=} in.B_1^1 \qquad B_1^1 \stackrel{\text{def}}{=} \overline{out}.B_0^1$$

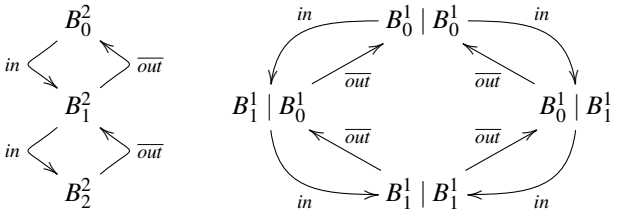
The corresponding LTS is



Analogously, for a buffer of capacity 2, one can 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 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 that 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.

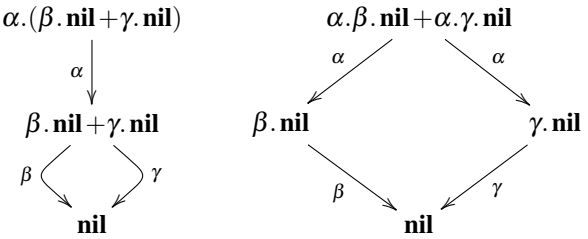
[2] A variant of trace equivalence, called completed trace semantics, is not prefix closed and will be discussed in Example 11.15.

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 at a later 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 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 $C[p]$. Consider how embarrassing the difference could be if α meant for a computer to ask the user whether 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 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 processes p and q again in Example 11.15, when discussing compositionality issues.

Given all the above, we can argue that neither graph isomorphism nor trace equivalence is a good candidate 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 Example 11.9 should not be considered behaviourally equivalent. Imagine that two opposing players are arguing about whether or not a system satisfies 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 respond. In the case of bisimulation, the system is composed of 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 what 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 (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 cannot simulate this move from $\beta.\mathbf{nil}$. Since Alice has a winning, two-move 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 are 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 *strongly 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 strongly 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 strongly 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 reflexive, symmetric and transitive pairs, unless needed.

In the following, when we 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 the *quotient set*.

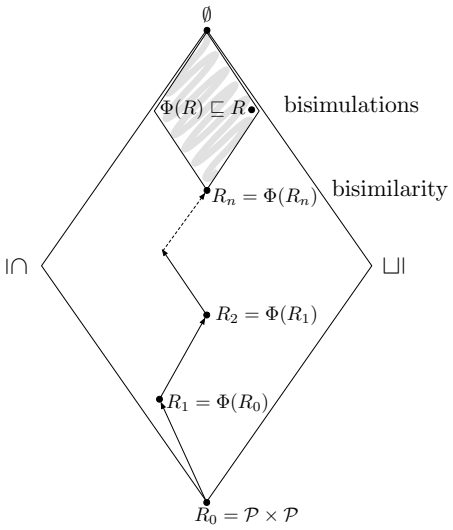


Fig. 11.10: The $CPO_\perp$ $(\wp(\mathscr{P} \times \mathscr{P}), \sqsubseteq)$

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 little 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 little 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 strongly 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 the definition of strong bisimulation. □

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 $s_1 \xrightarrow{\mu} s_1'$. Since $s_1\ \Phi(R_2)\ s_2$, there exists s_2' such that $s_2 \xrightarrow{\mu} s_2'$ and $s_1'\ R_2\ s_2'$. But since $R_2 \subseteq R_1$, we have $s_1'\ R_1\ s_2'$. Analogously for the case when $s_2 \xrightarrow{\mu} s_2'$. □

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

Example 11.12. To see an example of CCS processes p and q that are not strongly 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 $p_n \stackrel{\text{def}}{=} \underbrace{\tau.\ldots.\tau}_{n \text{ times}}.\mathbf{nil}$ 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 strongly 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 transition 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 that $s \simeq_n s$. 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 $p\ \Phi(\simeq_n)\ q$, i.e., $p \simeq_{n+1} q$.

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 proving 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 requiring 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 show 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 the 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 the 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 the inductive hypothesis. □

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

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

[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).

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 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. Recall 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 to 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 **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 that 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 strongly 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 $(D,\sqsubseteq)$ be 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, into 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 that $d_{min} \sqsubseteq f(d_{min})$.

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 by 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.

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

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

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

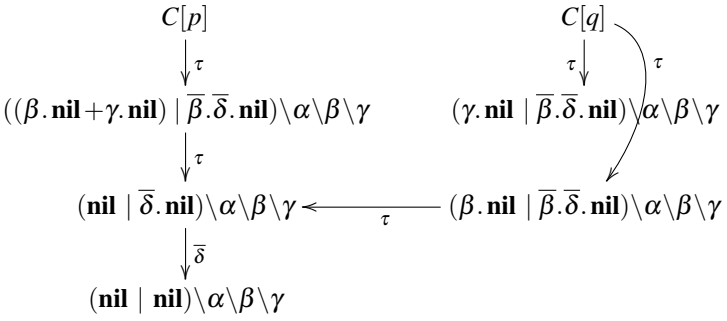
$$C[\cdot] \stackrel{\text{def}}{=} (\ \cdot \mid \overline{\alpha}.\overline{\beta}.\overline{\delta}.\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

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

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

Two processes are *completed trace equivalent* if they have the same traces and the same completed traces.

We know from Example 11.9 that p and q have the same traces. The completed traces of p are the same as those 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 guarantee the compositionality of CCS we must show that strong bisimilarity is a congruence relation with respect to all CCS contexts. To this aim, it is enough to prove that it is preserved by 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 us 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 that $(p_0 \mid p_1)\ R\ (q_0 \mid q_1)$. 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 $r_0 \mid r_1 \stackrel{\mu}{\to} r$. We need to prove that there exists s such that $s_0 \mid s_1 \stackrel{\mu}{\to} s$ 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 $r_0 \mid r_1 \stackrel{\mu}{\to} r$.

- The first case is when we have applied the first (Par) rule. So we have $r_0 \stackrel{\mu}{\to} r'_0$ 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, there exists s'_0 such that $s_0 \stackrel{\mu}{\to} s'_0$ and $(r'_0, s'_0) \in R_0$. Then, by applying the same inference rule we get $s_0 \mid s_1 \stackrel{\mu}{\to} s'_0 \mid s_1$. 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 $r_1 \stackrel{\mu}{\to} r'_1$ 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 $r_0 \stackrel{\lambda}{\to} r'_0$, $r_1 \stackrel{\overline{\lambda}}{\to} r'_1$, $\mu = \tau$ 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, there exists s'_0 such that $s_0 \stackrel{\lambda}{\to} s'_0$ and $(r'_0, s'_0) \in R_0$. Similarly, since $r_1\ R_1\ s_1$ and R_1 is a strong bisimulation relation, there exists s'_1 such that $s_1 \stackrel{\overline{\lambda}}{\to} s'_1$ and $(r'_1, s'_1) \in R_1$. Then, by applying the same inference rule we get $s_0 \mid s_1 \stackrel{\tau}{\to} s'_0 \mid s'_1$. 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 of Bisimilarity: Hennessy-Milner Logic

In this section we present a *modal logic* introduced by Matthew Hennessy and Robin Milner. Modal logic allows us to express concepts such 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 strongly 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 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.

$\Diamond_\mu F$: 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 $\langle\mu\rangle F$.

$\Box_\mu F$: 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.

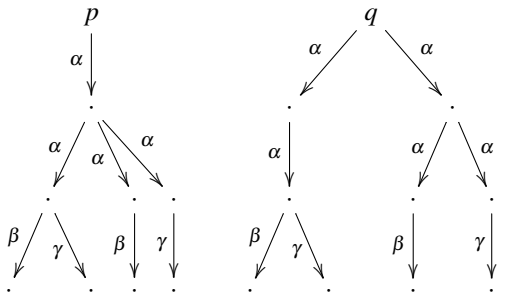
Notably, if p cannot execute any μ-transition, then $p \models \Box_\mu F$ for any formula F. For example, the formula $\Diamond_\alpha \mathit{true}$ is satisfied by all processes that can execute an α-transition, and the formula $\Box_\beta \mathit{false}$ is satisfied by all processes that cannot 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 work out which processes satisfy 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, the expressiveness of HM-logic is closed under negation, i.e., given 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:

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

Now we present two theorems which allow us to connect strong bisimilarity and modal logic. As we said, this connection is very important from both theoretical and practical points 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 a formula.

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 by $\mathscr{L}_k = \{F \in \mathscr{L} \mid \mathrm{md}(F) = k\}$.

The first theorem ensures that if two agents are not distinguished by the kth 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 the 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 in 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 be 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 strongly 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 strongly bisimilar, then there exists a formula F that 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 checking whether $q \models F$.

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 us 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, strongly 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 saying 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 strongly bisimilar, by transitivity and congruence of strong bisimilarity we have that p and q are strongly 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 any such agent *finite*. Finally, the axioms that characterise strong bisimilarity are the following:

$$p + \mathbf{nil} \simeq p$$
$$p_1 + p_2 \simeq p_2 + p_1$$
$$p_1 + (p_2 + p_3) \simeq (p_1 + p_2) + p_3$$
$$p + p \simeq p$$

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 strongly 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 strongly 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 strongly bisimilar, which can be readily done by exhibiting suitable strong bisimulation relations, similarly to the way we proved 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 strongly 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 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 nondeterminism. 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 strongly 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).

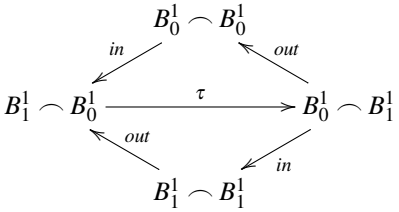
11.8 Weak Semantics of CCS

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 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 impossible 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 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 transition can involve several internal moves.

11.8.1 Weak Bisimilarity

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

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

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

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

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

As we did 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 that

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

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 the 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, which 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 nondeterministic choices. While quite unpleasant, the above fact is not in any way due to a CCS weakness, or misrepresentation of reality, but rather illuminates a general property of nondeterministic choice in systems represented as black boxes.

11.8.2 Weak Observational Congruence

As shown by 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 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 choice observable we can define the relation $\approx$ as follows.

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

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

Weak observational congruence can be defined directly by letting

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

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 $\approx$ is simply defined in terms of $\approx$: after the first step has been performed, other τ-labelled transitions can be simulated also by staying idle. Now it is obvious that $\alpha.\mathbf{nil} \not\approx \tau.\alpha.\mathbf{nil}$, because $\alpha.\mathbf{nil}$ cannot simulate the τ-transition $\tau.\alpha.\mathbf{nil} \xrightarrow{\tau} \alpha.\mathbf{nil}$.

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

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

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

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 to reach a state that is related by $\approx$ to p'. Thus $\approx$ is not a pre-fixpoint of Ψ.

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

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

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

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

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

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

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

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

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

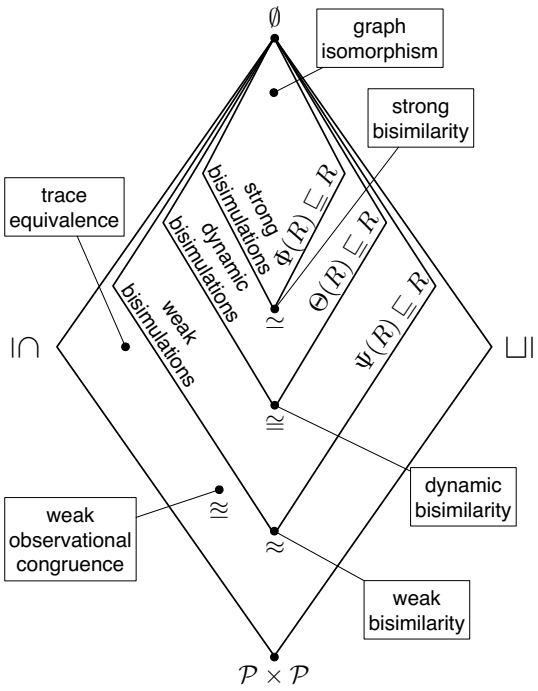


Fig. 11.11: Main relations in the $CPO_{\perp}$ $(\wp(\mathscr{P} \times \mathscr{P}), \sqsubseteq)$

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.

In all games, the rules for the attacker stay the same.

We recall 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 $\approxeq$ 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 $((p \mid q) \mid r)\backslash\alpha$ 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)

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

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 such as

$$\text{var } x$$

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 such as

$$x := v_i$$

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, such as $(c_1;c_2);c_3$, 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 strongly 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

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

Draw the LTS for r and s and prove that they are weakly observationally congruent by exploiting the axioms presented in Sections 11.7 and 11.8.2.

11.12. Consider the CCS agents

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

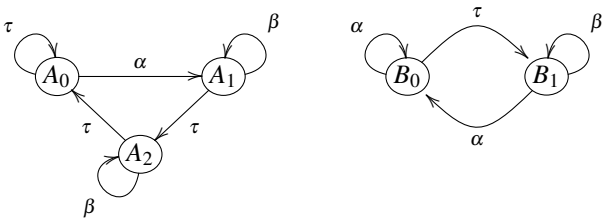
Prove that p and q are strongly bisimilar or exhibit an HM-logic formula F that can be used to distinguish them.

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 correspond to A_0 and B_0.
   Hint: Introduce a **rec** construct for each node in the diagram and name the process variables after 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 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 weakly observationally 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 us 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 consisting of 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 railway control systems to spacecraft 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-famous disaster stories of software failures[1] that (maybe) could have been avoided if formal methods had been employed, over the

[1] Top famous stories include the problems with the Therac-25 radiation therapy engine, which 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 of the Ariane 5.01 maiden flight due to an overflow in data conversion that caused a hardware exception and finally led to self-destruction.

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_12

years, formal methods have provided 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 and nuclear plant emergency systems.

Formal logics serve for writing down unambiguous specifications about how a program is supposed to behave and for reasoning about system correctness. Classically, we can divide the properties to be investigated into 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 possible to talk about concepts such as "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 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 they allow us not only to re-use algorithms for the verification of μ-calculus formulas to check whether temporal logic statements are satisfied, but also because temporal logic formulas are often more readable than specifications written in the μ-calculus.

12.2 Temporal Logic

Temporal logic has similarities to 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 us to look further than the "next" operator of HM-logic;
- 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 semantics view vs the bisimulation semantics view.

12.2.1 Linear Temporal Logic

In the case of *Linear Temporal Logic* (LTL), time is represented as a line. This means that the evolutions of the system are linear: they proceed from one 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 the *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 the *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 the *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 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 allow us 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 we did for the HM-logic, we define 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:

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

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 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:

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

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 different possible futures can be taken. CTL and CTL* extend LTL with two operators which allow us 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 operator 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 that $S, \pi^i \models \phi$
$S, \pi \models \phi_0\ U\ \phi_1$	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 appears 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, and greatest fixpoints with safety properties.

[3] An LTL formula ϕ is read as the CTL* formula $A\phi$. Namely, the structure where an 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): with each formula we associate the set of states of the transition system where the formula holds true. Then, the least and greatest fixpoint correspond quite nicely to the notion of the 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 with each proposition and with each variable a subset of states of the LTS. Then we define the denotational semantics of the 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 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 intersection 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 all have 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:

$$\begin{aligned}
S_0 &= \varnothing \\
S_1 &= \rho(p) \\
S_2 &= \rho(p) \cup \{v \mid \exists v' \in \rho(p).\ v \to v'\} \\
&= \{v \mid v \text{ can reach some } v' \in \rho(p) \text{ in no more than one step}\} \\
&\ldots \\
S_n &= \{v \mid v \text{ can reach some } v' \in \rho(p) \text{ in no more than } n-1 \text{ steps}\} \\
\bigcup_{n\in\mathbb{N}} S_n &= \{v \mid v \text{ has a finite path to some } v' \in \rho(p)\}
\end{aligned}$$

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 the μ-calculus is balanced by an equally great increase in awkwardness: we invite the reader to check for her/himself how relatively easy it 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 the 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:

$\mu x.\ (p \wedge \Diamond x) \vee q$:	corresponds to the CTL formula $E(p\ U\ q)$.
$\mu x.\ (p \wedge \Box x \wedge \Diamond x) \vee q$:	corresponds to the LTL/CTL formula $A(p\ U\ q)$. Note that the sub-formula $\Diamond x$ is needed to discard deadlock states.
$\nu x.\ \mu y.\ (p \wedge \Diamond x) \vee \Diamond y$:	corresponds to the CTL* formula $EGFp$: 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 the 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 of the exhaustive, possibly automatic, verification of whether a given model of a system meets a given logic specification of the properties the system should satisfy, such as 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 into 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 in 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 nonempty, 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 the 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 $\mathsf{NP} \cap \mathsf{coNP}$ (the 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 to 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 aforementioned, 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, for example 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 got 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 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 an LTL formula? Is it a CTL formula?

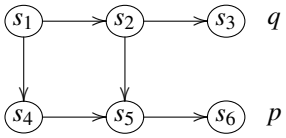
12.4. Prove that if the CTL* formula $AO\ \phi$ is satisfied, then also the formula $O\ A\ \phi$ is satisfied. Is the converse true?

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

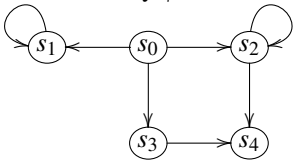
12.6. Given the μ-calculus formula

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

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



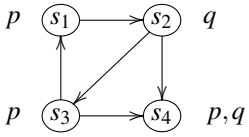
12.7. Given the μ-calculus formula $\phi \stackrel{\text{def}}{=} \nu x.\ \Diamond x$, compute its denotational semantics, spelling out what are the states that satisfy ϕ, and evaluate it on the LTS below:



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

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

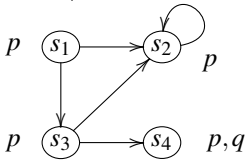
Write the denotational semantics of ϕ and evaluate it on the LTS below:



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 on the LTS below:

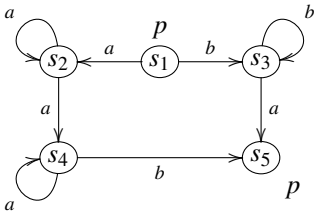


12.10. Let us extend the μ-calculus with the formulas $\langle A \rangle \phi$ and $[A]\phi$, where A is a set of labels: they represent, respectively, the ability to perform a transition with some label $a \in A$ and reach a state that satisfies ϕ, and the necessity to reach a state that satisfies ϕ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

1. Define the semantics $[\![\langle A \rangle \phi]\!] \rho$ and $[\![[A]\phi]\!] \rho$.
2. When $A = \{a_1, ..., a_n\}$, let us write $\langle a_1, ..., a_n \rangle \phi$ and $[a_1, ..., a_n]\phi$ in place of $\langle A \rangle \phi$ and $[A]\phi$, 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 the π-calculus. It is not an exaggeration to affirm that the π-calculus plays for reactive systems the same foundational role that the 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 to this feature is *name mobility*. The operational semantics of the 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 us 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

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_13

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, for example

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

for some p'; 2) q waits for a message on a, for example

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

for some q' that can exploit x; and 3) r wants to input a channel name on b, which it will use to send a message m, for example

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

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

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

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, for instance

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

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

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

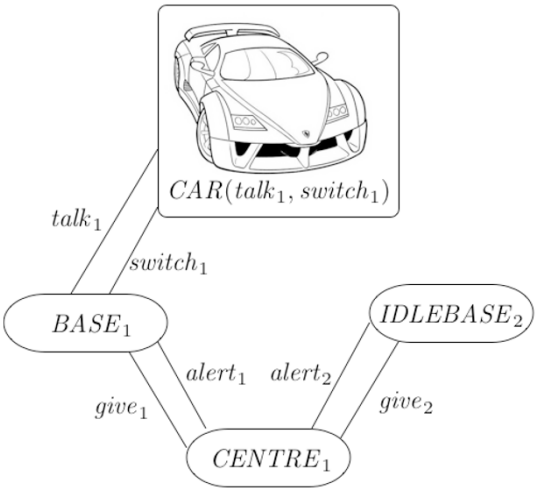
and after the communication it becomes

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

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 that 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 car 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 $switch_i$ to allow 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 $\overline{switch_1}(talk_2, switch_2)$. 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 links 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 to the environment and 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 she 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 S to serve multiple requests from Alice by issuing multiple instances of the server process. Bob waits to receive a name to replace 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 the 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 nondeterministic 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 analogous to 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 binding operators of the 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 denote the fact that 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 private name 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 $\mathrm{bn}(\cdot)$ is 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 a process to perform invisible actions.

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

As we said, 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 a process to receive as 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 $z \in \mathrm{fn}((y)\bar{y}z.\,\mathbf{nil})$)

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

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 a process 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 $login(y).[y=pwd]p$.

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 p can perform and that does not involve y can be performed by $(y)p$.

13.3.7 Scope Extrusion

Now we present the most important rules of the 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), like 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 into 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 us to replicate a process as many times as needed, in a recursive fashion, without consuming it. Notice that $!p$ is also able to perform synchronisations between two copies of p, if it is 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:

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

where p, q, r are π-calculus processes. The process $(y)\overline{x}\,y.p$ would like to set up a private channel with $x(z).r$, which however should remain hidden from 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:

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

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

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

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 in the 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 the 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 applying basic reduction rules. In fact it can be proved that for each π-calculus process p there exist

- 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 $p \mapsto q$ to indicate that p reduces to q in one step. The rules defining the relation $\mapsto$ are the following:

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

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

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 nondeterministic 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 us 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 the 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 the *strong bisimulation* of the 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 the 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 shortly, the second option leads to a finer semantics. Finally, we will present the *weak bisimulation* for the 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 the other four cases are present, where q challenges p to (bi)simulate its transitions. Note that in the case of the 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 for all possible received names w we are able to show $p'[{}^{w}/_{y}]\,S\,q'[{}^{w}/_{y}]$ for suitable q'. Notice that also names w 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:

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

[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 $w \in \mathrm{fn}(p') \cup fn(q')$ and for a *single* fresh name $w \notin \mathrm{fn}(p') \cup fn(q')$, 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 a 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 13.4. 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 the received value $w=z$ but not in the other cases. On the other hand, if the right choice is selected, then **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 into 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 in 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 might expect, weak (early and late) bisimilarities are not congruences due to aliasing, as was already the case for strong bisimilarities. In addition, weak (early and late) bisimilarities are not congruences for a choice context, as 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 $\overline{x}\langle y\rangle$ 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 more than one name to be communicated in a single action:

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

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 $\overline{x}\langle P\rangle.p$ 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 $\overline{x}y.p$ it must be that $p = \mathbf{nil}$.

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 $p' \xrightarrow{\alpha} q'$ 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. Also modify the input clause for 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 whether 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 nondeterminism together with a specification logic. Chapter 16 defines the syntax and 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 The future is largely unpredictable. Nondeterminism allows modelling of 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 *probabilistic* 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 the *Markov property* and it allows us 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 nondeterminism to represent choices and parallelism. Probability can be viewed as a refinement of nondeterminism, 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 with 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, nondeterminism 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 with each event. Often in stochastic systems there is no explicit nondeterministic choice: when a race between events is enabled, the fastest operation is chosen.

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_14

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 fact that the probability of evolving 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 us 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 allows modelling of 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 we are not 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 events, 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 a *measurable space*. We are interested in 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 heads (H) and tails (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.

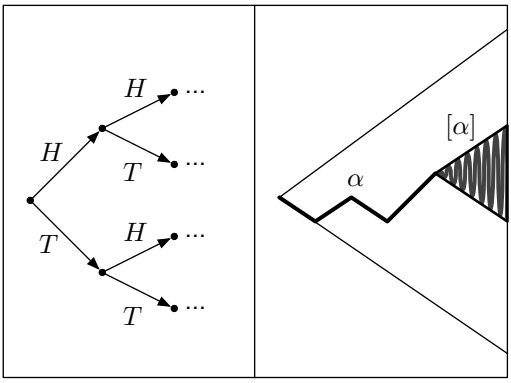


Fig. 14.1: The shadow of α

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, recall that the definition of σ-field ensures that the 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 the Scott topology, associated with 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 corresponding Borel σ-field; then associating each open interval with 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* with 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 with 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 this with 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 heads and 1 to tails 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 countable 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 to have 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, and $X : \Omega \to \mathbb{R}$ be a continuous random variable over S. We call the *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 that describes the relative probability of a continuous random variable taking 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

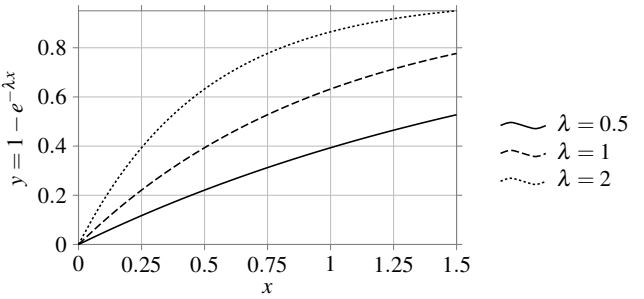


Fig. 14.2: Exponential probability laws with different rates λ

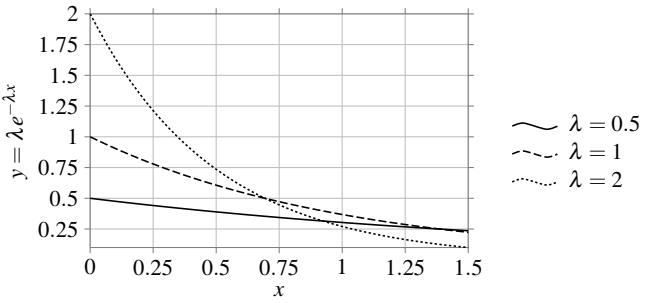


Fig. 14.3: Exponential density distributions with different rates λ

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

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 the *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:

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

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 Figures 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 lose 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 has not decayed yet at time t_0 is the same for any choice of t_0, as it depends only 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 one event will be executed before another. This corresponds in our view to calculating 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 rates 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 some later 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 elements 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 made 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 processes are totally independent of the past. *Markov chains* extend the classic theory by dealing with processes where the value of each variable is influenced by the previous value. This means that in Markov processes the next outcome of the system is influenced only by the previous state. One might want 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 the *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 over time. We will consider only the simpler case of homogeneous Markov chains, gaining the possibility to shift the time axis backward 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 Chains

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 chains* (DTMCs), and those in which $T = \mathbb{R}$, referred to as *continuous time Markov chains* (CTMCs).

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 $x, x_0, x_1, ..., x_n \in \mathbb{R}$

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

Since we are restricting our attention to homogeneous chains, 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 us 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 $P(X_t = j \mid X_0 = i) = 1 - e^{-\lambda_{i,j}t}$.

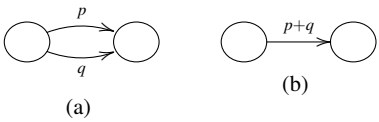
We remark that the exponential random variable is the only continuous random variable with the memoryless property, i.e., CTMCs are necessarily exponentially distributed.

14.4.2 DTMCs as LTSs

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

A difference between LTSs and PTSs is that in LTSs 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 PTSs we cannot have this kind of situation since two different transitions between the same pair of states have the same meaning as a single transition labelled 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 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 by a matrix P whose indices i, j represent states s_i, s_j and each element $a_{i,j}$ is the probability that given that the system is in the state i it will 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 relying 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 state s_i at 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 by 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 reasons.

For some special classes 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).

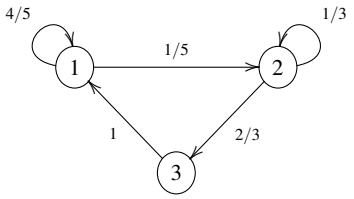


Fig. 14.5: A DTMC

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 state 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 a 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 every 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 of 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 with the chain, under the additional constraint that the sum of all probabilities in π 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 of being in the state i is given by the sum of the probabilities of being in any other state j weighted by the probability of moving 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 of 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 CTMCs as LTSs

Continuous time Markov chains can also 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, as for DTMCs, we have that two different transitions between the same two states are merged into 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 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 this 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 $\{X_t\}$ be 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 an *infinitesimal generator*. This definition is convenient for steady state analysis, as explained at the end of the next section.

14.4.5 Embedded DTMC of a CTMC

Often the study of a CTMC is very hard, particularly in terms of computational complexity. So it is useful to have a standard way to discretise the CTMC by synthesising a DTMC, called the *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 the rates to 1 in order to produce a probability distribution.

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 might think of modifying 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 an 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 class induced by R. By the definition of bisimulation we have that given 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 $\Phi : \wp(S \times S) \rightarrow \wp(S \times S)$ defined by letting

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

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 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 transition 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

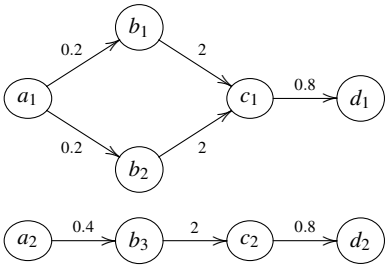


Fig. 14.6: CTMC bisimilarity

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 $\Phi_C : \wp(S \times S) \to \wp(S \times S)$ 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 from s_2.

Definition 14.24 (CTMC bisimilarity $\simeq_C$). 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 might think that the same argument about bisimilarity that we have exploited for CTMCs could also be 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 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 $\Phi_D : \wp(S \times S) \to \wp(S \times S)$ 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 bisimilarity $\simeq_D$). 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

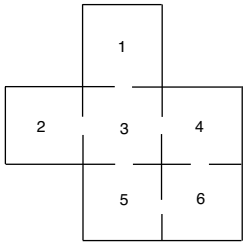
$$\forall I.\ \gamma_D(s_1)(I) = * \neq \gamma_D(s)(I) \in \mathbb{R}$$

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).

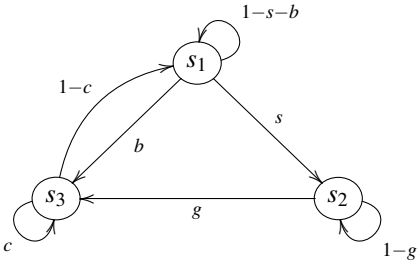
1. Draw the transition graph and give the matrix P for this DTMC.
2. Show that it is ergodic and compute the steady state distribution.
3. Assuming the mouse is initially in room 1, what is the probability that it is in room 6 after three steps?

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 this is so.

14.4. With the Markov chain below we intend to represent the scenario where Mario, a taxi driver, is looking for customers. In state s_1, Mario is parked waiting for customers, who 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, to move around (state s_2), driving in the busiest streets of town looking for clients, who 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 customers 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 of breaking down is $\frac{1}{20} = 0.05$ and the probability of finishing the task (and going to waiting) is $\frac{1}{10} = 0.1$.
- If the machine is under repair there is a $\frac{1}{10} = 0.1$ probability of getting 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 of getting 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 that it is 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 spawned with rate

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

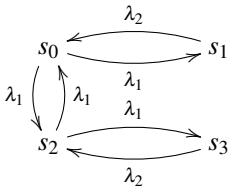
for some fixed λ, i.e., the rate decreases as the number of instances already running increases,[3] while an instance is terminated with rate

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

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.
4. Consider a second system composed of N independent components, each with two states and transitions with rates λ and μ to change state. Are the two systems CTMC bisimilar?

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 any 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 clients 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 Nondeterminism

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 nondeterminism, 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 can affect its future state (its reaction). This is the reason why this type of model is called "reactive". Formally, we have that a *reactive probabilistic transition system* (also called a *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".

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_15

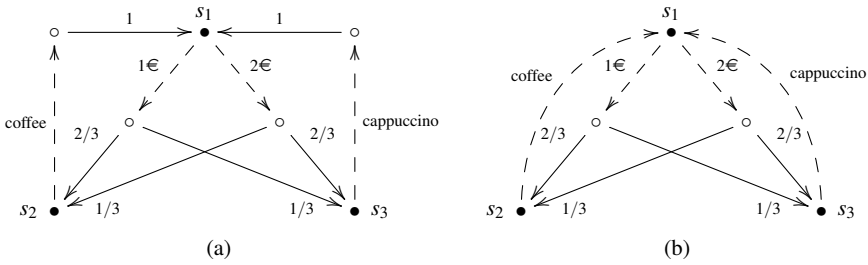


Fig. 15.1: A reactive PTS representing a coffee maker

where we recall that S is the set of states, $1 = \{*\}$ is a singleton used to represent the deadlock states, and $\mathscr{D}(S)$ 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, given a non-deadlock source state and an action, the next state probabilities must sum to 1, while in a generative system, given a 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 than 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 by 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 cappuccino, which 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 $\alpha_r : S \to L \to (\mathscr{D}(S) \cup 1)$. Let us define the function $\gamma_r : S \to L \to \wp(S) \to \mathbb{R}$ as follows:

$$\gamma_r(s)(\ell)(I) = \begin{cases} 0 & \text{if } \alpha_r(s)(\ell) = * \\ \sum_{s' \in I} \alpha_r(s)(\ell)(s') & \text{otherwise} \end{cases}$$

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

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

Definition 15.1 (Reactive bisimilarity $\simeq_r$). A *reactive bisimulation* is a relation R such that $R \subseteq \Phi_r(R)$ and the *reactive bisimilarity* $\simeq_r$ is the relation

$$\simeq_r \stackrel{\text{def}}{=} \bigcup_{R \subseteq \Phi_r(R)} R$$

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 into 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.

As for HM-logic, the equivalence induced by Larsen-Skou logic formulas coincides 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 characterisation). *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 Nondeterminism

In this section we add nondeterminism to generative and reactive systems. Correspondingly, we introduce two classes of models called *Segala automata* and *simple Segala automata*, named after Roberto Segala who developed them in 1995. In both cases we use nondeterminism to allow the system to choose between different probability distributions.

15.3.1 Segala Automata

Segala automata are generative systems that combine probability and nondeterminism. When the system has to move from a state to another, first of all it has to nondeterministically choose 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 there 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 explicitly model deadlock states, because we can use the empty set for this purpose.

Example 15.3 (Segala automata). Let us consider the system in Figure 15.2. We have an automaton 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 nondeterministically (dashed arrows) between two different distributions d_1 and d_2:

$$\alpha_s(s_1) = \{d_1, d_2\} \quad \text{where} \begin{cases} d_1(\text{flip}, s_2) = 1/2 & d_1(\text{flop}, s_3) = 1/2 \\ d_2(\text{flip}, s_2) = 2/3 & d_2(\text{flop}, s_3) = 1/3 \end{cases}$$

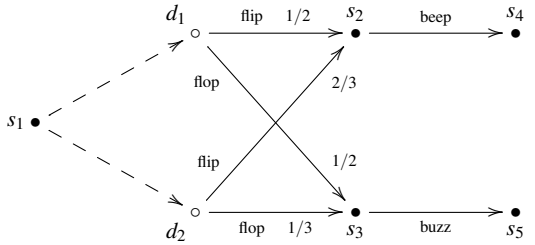


Fig. 15.2: A Segala automaton

(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 nondeterminism 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 automaton). 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

$$\alpha_{\mathsf{simS}}(s_1) = \{(a,d_1),(c,d_2),(a,d_3)\} \text{ where } \begin{cases} d_1(s_2) = 1/2 & d_1(s_3) = 1/2 \\ d_2(s_4) = 1/3 & d_2(s_5) = 2/3 \\ d_3(s_1) = \varepsilon & d_3(s_6) = 1-\varepsilon \end{cases}$$

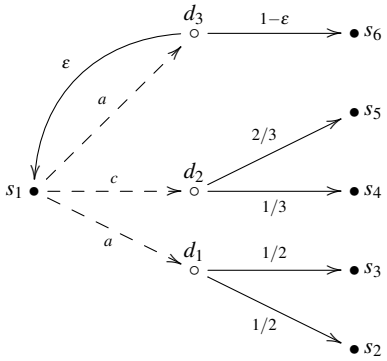


Fig. 15.3: A simple Segala automaton

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 Nondeterminism, Probability and Actions

As we have seen, there are many ways to combine probability, nondeterminism 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 nondeterminism. 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 system we can alternate probabilistic and nondeterministic choices and can partition the states accordingly. (Again, a state s is deadlock when $\alpha_{\mathsf{a}}(s) = \varnothing$.)

The second type of system that we present is that of *bundle transition systems*. In this case the system associates a distribution with subsets of nondeterministic choices. Formally, the transition function has the form

$$\alpha_{\mathsf{b}} : S \to \mathscr{D}(\mathscr{P}(L \times S))$$

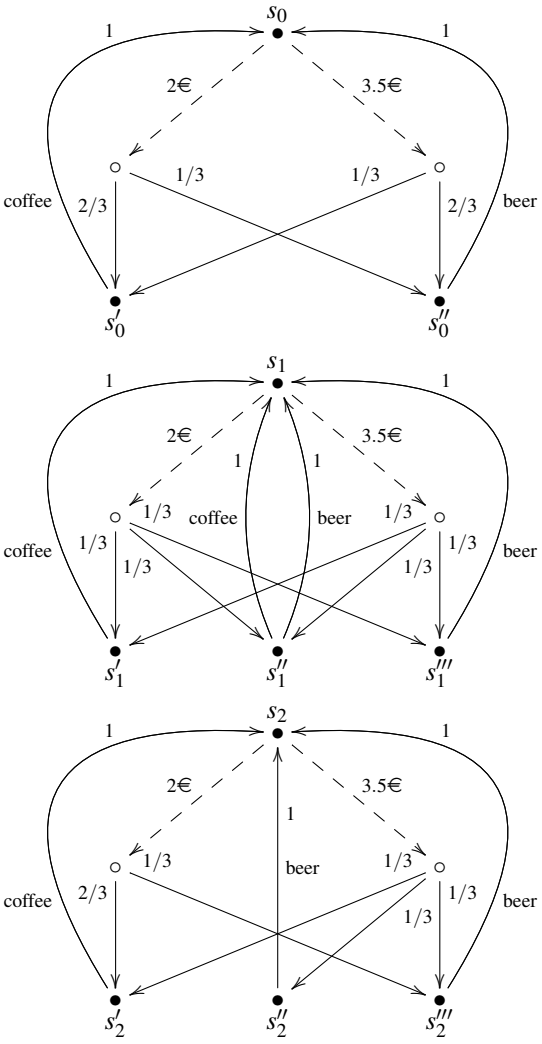
So when a bundle transition system has to perform a transition, first of all it uses a probability distribution to choose a set of possible choices, then it nondeterministically picks one of these.

Problems

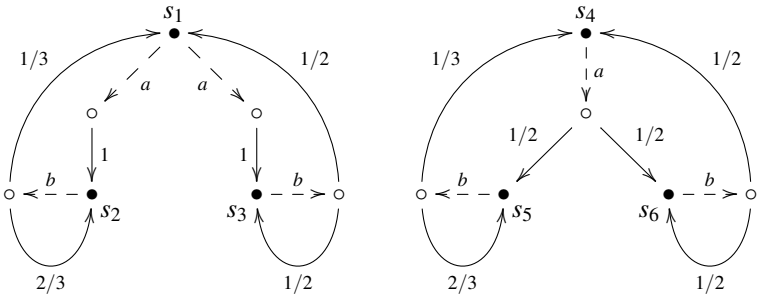
15.1. In what sense is a Segala automaton the most general model?

1. Show in what 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 system (PLTS) be a reactive system 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* PLTSs.
3. Furthermore, consider the non-stopping subclass of generative PLTSs and show an example where some states are not bisimilar.
4. Moreover, give the definition of bisimilarity also for Segala PLTSs, 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 carrying out their quantitative analysis. PEPA builds on CSP (Communicating 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) communication. PEPA actions are labelled with rates and without much effort a CTMC can be derived from the LTS of a PEPA process 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 such as:

- 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 such as:

- 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.

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7_16

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 less modular approaches;
formality:	a rigorous semantics giving a precise meaning to all terms in the language and resolving 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 on 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 offers abstraction mechanisms.

PEPA was obtained by extending CSP (Communicating 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 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 the *cooperation combinator*. More precisely, it denotes an indexed family of operators, one for each possible set of actions L. The set L is called the *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 with 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 a *constant*, of a recursively defined process that we assume to be 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 the 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 evolves in one step 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 $P\backslash\alpha$ behave 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 $\underset{L}{\bowtie}$: 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 no explicit probabilistic operators in PEPA, but probabilistic behaviour is obtained by associating an exponentially distributed continuous random variable with 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 outside the control of the 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 as 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 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:

$$\begin{aligned} Appl \stackrel{\text{def}}{=} & (think, p_1 \cdot \lambda).(local, m).Appl \\ & + (think, p_2 \cdot \lambda).(request, rq).(respond, rp).Appl \end{aligned}$$

$P \underset{L}{\bowtie} Q$: has the same meaning as 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 as 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* from 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 to be available in a separate set Δ of declarations. Using recursive definitions like 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 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:

$$\begin{array}{lcl} P, Q & ::= & S \quad | \quad P \underset{L}{\bowtie} Q \quad | \quad P/L \\ S, T & ::= & (\alpha, r).S \quad | \quad S + T \quad | \quad C \end{array}$$

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 that $(C \stackrel{\text{def}}{=} S) \in \Delta$ 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 that transition labels also carry 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 the *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 slower 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, as choices are independent, 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 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

$$m \times n \times \frac{r}{m} = n \times r = r_\alpha(P \underset{\{\alpha\}}{\bowtie} Q)$$

Example 16.2. Let us consider the PEPA process $S \stackrel{\text{def}}{=} C_1 \underset{\{\alpha\}}{\bowtie} C_2$, where

$$C_1 \stackrel{\text{def}}{=} (\alpha, r_1).C_1 + (\beta, r_2).C_1 \qquad C_2 \stackrel{\text{def}}{=} (\alpha, \top).C2 + (\beta, r_2).C_2$$

The corresponding LTS has one state and two (self looping) transitions with labels (α, r_1) and $(\beta, 2r_2)$: the former models the cooperation between the two components C_1 and C_2 on the shared action α, while the latter models the two possible ways of executing the action β. In the first case, the rate $\top$ associated with C_2 does not influence the observable rate of α in S. In the second case, the apparent rate of action β is the sum of the rates for β in C_1 and in C_2.

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 several transitions are 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 the 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 the 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 16.3 (Roland the gunman). We want to model a Far West duel. We have two main characters: Roland the gunman and his enemies. Upon his 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 gunfight. 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 guns

$$Roland_{\text{idle}} \stackrel{\text{def}}{=} (\text{attack}, \top).Roland_2$$

In front of his enemy, Roland can act in three ways: if Roland hits the enemy then he reloads his gun and returns to 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 Roland's 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.

$$Roland_{\text{dead}} \stackrel{\text{def}}{=} \textbf{nil}$$

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_{fire}	$= 1$	one shot per second
$r_{\text{hit-success}}$	$= 0.8$	80% chance of success
r_{hit}	$= 0.8$	$r_{\text{fire}} \times r_{\text{hit-success}}$
r_{miss}	$= 0.2$	$r_{\text{fire}} \times (1 - r_{\text{hit-success}})$
r_{reload}	$= 0.\overline{3}$	3 seconds to reload
r_{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 at least 10 seconds passing from the first attack on Roland until the time he has hit three 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

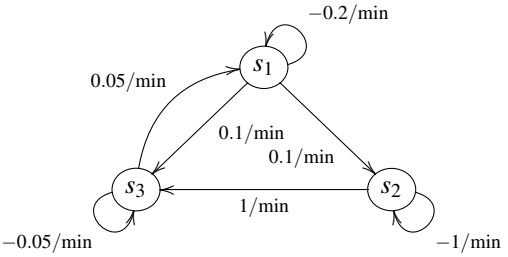
16.1. We have defined CTMC bisimilarity in the case of *unlabelled* transition systems, while the PEPA transition system is labelled. Extend the definition of bisimilarity to the labelled version.

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 of 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.

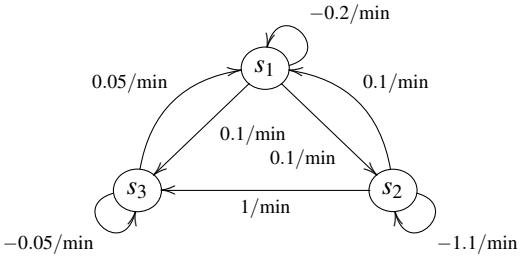
16.3. In a multiprocessor system with shared memory, processes must compete to use the memory bus. Consider the case of two identical processes. Each process has cyclic behaviour: it performs some local activity (local action *think*), accesses the bus (synchronisation action *get*), operates on the memory (local action *use*) and then releases the bus (synchronisation action *rel*). The bus has cyclic behaviour 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 14.4, but this time represented as the CTMC in the figure below, where rates are defined in 1/minutes, e.g., customers 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 DTMCs: $\pi\, Q = 0$ 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 customers) 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 DTMCs: $\pi\, Q = 0$ 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] Recall 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.

16.7. 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. 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 is the probability that the system will still be 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 DTMCs.

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 at T_0 and composing in parallel all of them with $\bowtie_L$, with L being the set of synchronised actions;
3. draw the LTS for the case $n = 2$, and compute the bisimilarity relation;
4. 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 nonempty sequences of $\mathtt{b}$s. The strings in L_A are all nonempty 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:
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{b} \in L_B}(3)}{\mathtt{b}\,\mathtt{b} \in L_B}(4)}{\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(2)}{\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(1)}{\mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(1)$$
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

R. Bruni and U. Montanari, *Models of Computation*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-319-42900-7

$s' \in L_B$ (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 $odd(x)$ is 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

$$\begin{array}{c} \langle w, \sigma\rangle \to \sigma' \;\nwarrow_{\sigma' = \sigma}\; \langle y \geq 0, \sigma\rangle \to \textbf{false} \\ \nwarrow^{*} \quad \square \end{array}$$

If instead $\sigma(y) > 0$, we have

$$\begin{array}{rll} \langle w, \sigma\rangle \to \sigma' & \nwarrow & \langle y \geq 0, \sigma\rangle \to \textbf{true}, \langle c, \sigma\rangle \to \sigma'', \langle w, \sigma''\rangle \to \sigma' \\ & \nwarrow^{*} & \langle c, \sigma\rangle \to \sigma'', \langle w, \sigma''\rangle \to \sigma' \\ & \nwarrow^{*} & \langle \textbf{skip}, \sigma\rangle \to \sigma'', \langle w, \sigma''\rangle \to \sigma' \\ & \nwarrow^{*}_{\sigma'' = \sigma} & \langle w, \sigma\rangle \to \sigma' \end{array}$$

Since we reach the same goal from which we started, the command diverges.
Finally, if instead $\sigma(y) = 0$, we have

$$\begin{array}{rl}
\langle w,\sigma\rangle \to \sigma' & \nwarrow \quad \langle y \geq 0,\sigma\rangle \to \textbf{true}, \langle c,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\
& \nwarrow^* \quad \langle c,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\
& \nwarrow^* \quad \langle y := y+1,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\
& \nwarrow^*_{\sigma''=\sigma[^1/_y]} \langle w,\sigma[^1/_y]\rangle \to \sigma'
\end{array}$$

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 $\langle b,\sigma\rangle \to \textbf{false}$, we have

$$\begin{array}{rl}
\langle c_1,\sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} \langle b,\sigma\rangle \to \textbf{false} \\
& \nwarrow^* \quad \square \\
\langle c_2,\sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} \langle b,\sigma\rangle \to \textbf{false} \\
& \nwarrow^* \quad \square
\end{array}$$

Thus it must be $\langle b,\sigma\rangle \to \textbf{true}$. In this case, we have

$$\begin{array}{rl}
\langle c_1,\sigma\rangle \to \sigma' & \nwarrow \langle b,\sigma\rangle \to \textbf{true}, \langle c,\sigma\rangle \to \sigma'', \langle c_1,\sigma''\rangle \to \sigma' \\
& \nwarrow^* \langle c,\sigma\rangle \to \sigma'', \langle c_1,\sigma''\rangle \to \sigma' \\
\langle c_2,\sigma\rangle \to \sigma' & \nwarrow \langle b,\sigma\rangle \to \textbf{true}, \langle c',\sigma\rangle \to \sigma'', \langle c_2,\sigma''\rangle \to \sigma' \\
& \nwarrow^* \langle c',\sigma\rangle \to \sigma'', \langle c_2,\sigma''\rangle \to \sigma' \\
& \nwarrow \langle b,\sigma\rangle \to \textbf{true}, \langle c,\sigma\rangle \to \sigma'', \langle c_2,\sigma''\rangle \to \sigma' \\
& \nwarrow^* \langle c,\sigma\rangle \to \sigma'', \langle c_2,\sigma''\rangle \to \sigma'
\end{array}$$

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 $P(\langle \textbf{skip}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{skip}).\ \sigma(y) = \sigma(y)$, which 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 $\sigma(y) = \sigma'(y)$.

ifft: 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

$$\langle x \neq 0, \sigma\rangle \to \textbf{true} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle w, \sigma''\rangle \to \sigma'$$
$$P(\langle w, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma''(x) \geq 0 \ \wedge\ \sigma' = \sigma''\left[{}^{\sigma''(x)+\sigma''(y)}/_{y}, {}^{0}/_{x}\right]$$

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 the inductive hypothesis we have $\sigma''(x) \geq 0$, thus $\sigma(x) \geq 1$ and hence $\sigma(x) \geq 0$. Moreover, by the 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 implications 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 S is, the smaller $\wp(\mathbb{N}) \setminus S$ is 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.
Let us now address continuity of f and g.
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 continuous).

2. 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, this 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 was 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 exist, 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 be such that $\beta = \alpha\gamma$ and $\alpha = \beta\delta$, then $\alpha = \alpha\gamma\delta$: if $\alpha \in V^*$, then it must be that $\gamma = \delta = \varepsilon$ and $\alpha = \beta$; if $\alpha \in V^\infty$, from $\beta = \alpha\gamma$ it follows that $\beta = \alpha$;
transitive:	we assume $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \gamma$ and we need to prove that $\alpha \sqsubseteq \gamma$; let δ and ω be 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 that $\alpha_i \in V^*$ 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, taking $\alpha \in V^\infty$ 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

$$\begin{array}{rl}
\langle c, \sigma\rangle \to \sigma' & \nwarrow\ \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c'', \sigma''\rangle \to \sigma' \\
\nwarrow *_{\sigma''=\sigma[^0/_x]} & \langle x = 0, \sigma[^0/_x]\rangle \to \textbf{true}, \quad \langle c_1, \sigma[^0/_x]\rangle \to \sigma' \\
\nwarrow^* & \langle c_1, \sigma[^0/_x]\rangle \to \sigma' \\
\langle c', \sigma\rangle \to \sigma' & \nwarrow\ \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma' \\
\nwarrow *_{\sigma''=\sigma[^0/_x]} & \langle c_1, \sigma[^0/_x]\rangle \to \sigma'
\end{array}$$

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

$$\begin{aligned}
\Gamma_{b,c}\ \varphi\ \sigma &= \mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}\,[\![c]\!]\,\sigma), \sigma \\
\Gamma_{b,c'}\ \varphi\ \sigma &= \mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}\,[\![c']\!]\,\sigma), \sigma \\
&= \mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![c]\!]\,\sigma, \mathscr{C}\,[\![\textbf{skip}]\!]\,\sigma), \sigma \\
&= \mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![c]\!]\,\sigma, \sigma), \sigma
\end{aligned}$$

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 already be 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 correctness 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 $\langle b, \sigma' \rangle \to \mathbf{false}$ and $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$. We need to prove that

$$P\left(\langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma \rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma = \sigma'$$

From $\langle b, \sigma' \rangle \to \mathbf{false}$ it follows $\mathscr{B}[\![b]\!](\sigma') = \mathbf{false}$. We have

$$\begin{aligned}
\mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma &\stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to^* \sigma, \mathscr{C}[\![c]\!]\sigma \\
&= \mathscr{B}[\![b]\!]^*\sigma' \to^* \sigma, \sigma' \\
&= \mathscr{B}[\![b]\!]\sigma' \to^* \sigma, \sigma' \\
&= \mathbf{false} \to^* \sigma, \sigma' \\
&= \mathbf{false} \to \sigma, \sigma' \\
&= \sigma'
\end{aligned}$$

undo: we assume that $\langle b, \sigma' \rangle \to \mathbf{true}$ and $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$. We need to prove that

$$P(\langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma \rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma = \sigma$$

From $\langle b, \sigma' \rangle \to \mathbf{true}$ it follows that $\mathscr{B}[\![b]\!](\sigma') = \mathbf{true}$. We have

$$\begin{aligned}
\mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma &\stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to^* \sigma, \mathscr{C}[\![c]\!]\sigma \\
&= \mathscr{B}[\![b]\!]^*\sigma' \to^* \sigma, \sigma' \\
&= \mathscr{B}[\![b]\!]\sigma' \to^* \sigma, \sigma' \\
&= \mathbf{true} \to^* \sigma, \sigma' \\
&= \mathbf{true} \to \sigma, \sigma' \\
&= \sigma
\end{aligned}$$

Finally, we extend the proof of completeness 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

$$P(\mathbf{do}\ c\ \mathbf{undoif}\ b) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma = \sigma' \Rightarrow \langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma \rangle \to \sigma'$$

Let us take σ and σ' such that $\mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma = \sigma'$. We need to prove that $\langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma \rangle \to \sigma'$. Since $\mathscr{C}[\![\mathbf{do}\ c\ \mathbf{undoif}\ b]\!]\sigma = \sigma'$ it must be that $\mathscr{C}[\![c]\!]\sigma = \sigma''$ for some $\sigma'' \neq \bot_{\Sigma_\bot}$ and by the inductive hypothesis $\langle c, \sigma \rangle \to \sigma''$. We distinguish two cases.

$\mathscr{B}[\![b]\!]\sigma'' = \mathbf{false}$: then $\sigma' = \sigma''$ and $\langle b, \sigma'' \rangle \to \mathbf{false}$. Since $\langle c, \sigma \rangle \to \sigma''$ we apply rule (do) to derive $\langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma \rangle \to \sigma'' = \sigma'$.

$\mathscr{B}[\![b]\!]\,\sigma'' = \mathbf{true}$: 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

$$\mathbf{rec}\ \underbrace{f}_{\tau_2\to int}.\ \lambda\ \underbrace{x}_{\tau*int}.\ \mathbf{if}\ \mathbf{snd}(\underbrace{\underbrace{x}_{\tau*int}}_{int})\ \mathbf{then}\ \underbrace{1}_{int}\ \mathbf{else}\ \underbrace{\underbrace{f}_{\tau_2\to int}\ (\underbrace{\mathbf{fst}(x)}_{int\to\tau_1},\ (\underbrace{\underbrace{\mathbf{fst}(x)}_{\tau=int\to\tau_1}\ \underbrace{\mathbf{snd}(x)}_{int}}_{\tau_1}))}_{\tau_2=(int\to\tau_1)*\tau_1}}_{int}}_{(\tau*int)\to int}$$

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 that $\tau = (int \to \tau_1)$ 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

$$t \stackrel{\text{def}}{=} \lambda\underbrace{\ell}_{\tau}.\ \mathbf{fst}(\mathbf{snd}(\ \underbrace{\mathbf{snd}(\underbrace{\ell}_{\tau})}_{int*(int*int)}\ ))_{\tau\to int}$$

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

$$\underset{int\to\tau}{\underline{\ f\ }}\ .\ \lambda\ \underset{\tau*int}{\underline{\ x\ }}\ .\ \textbf{if}\ \textbf{snd}(\underbrace{\underset{\tau*int}{\underline{\ x\ }}}_{int})\ \textbf{then}\ \underbrace{\textbf{fst}(x)}_{\tau}\ \textbf{else}\ \underbrace{\underset{int\to\tau}{\underline{\ f\ }}\ \underbrace{(\textbf{snd}(x))}_{int}}_{\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{(\tau*int)\to\tau}$$

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

$$t_1 \stackrel{\text{def}}{=} \lambda\ \underset{int}{\underline{\ x\ }}\ .\ \lambda\ \underset{\tau_1}{\underline{\ y\ }}\ .\ \underbrace{\underset{int}{\underline{\ x\ }} + \underset{int}{\underline{3}}}_{int}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad}_{\tau_1\to int}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad}_{int\to\tau_1\to int}$$

$$t_2 \stackrel{\text{def}}{=} \lambda\ \underset{int*\tau_2}{\underline{\ z\ }}\ .\ \underbrace{\textbf{fst}(\underset{int*\tau_2}{\underline{\ z\ }})}_{int} + \underset{int}{\underline{3}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad}_{int}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad}_{(int*\tau_2)\to int}$$

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:

$$\begin{array}{rl}
((t_1\ 1)\ c) \to c_1 & \nwarrow\ (t_1\ 1) \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1 \\
& \nwarrow\ t_1 \to \lambda x'.\ t'',\quad t''[^{1}/_{x'}] \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1 \\
\nwarrow_{x'=x,\ t''=\lambda y.\ x+3} & \lambda y.\ 1+3 \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1 \\
\nwarrow_{y'=y,\ t'=1+3} & 1+3 \to c_1 \\
\nwarrow_{c_1=n_1 \underline{+} n_2} & 1 \to n_1,\quad 3 \to n_2 \\
\nwarrow^{*}_{n_1=1,\ n_2=3} & \square
\end{array}$$

Thus $c_1 = n_1 \underline{+} n_2 = 1 \underline{+} 3 = 4$ is the canonical form of $((t_1\ 1)\ c)$.

$$\begin{array}{rl}
(t_2\ (1,c)) \to c_2 & \nwarrow\ t_2 \to \lambda z'.\ t',\quad t'[^{(1,c)}/_{z'}] \to c_2 \\
\nwarrow_{z'=z,\ t'=\textbf{fst}(z)+3} & \textbf{fst}((1,c)) + 3 \to c_2 \\
\nwarrow_{c_2=n_1 \underline{+} n_2} & \textbf{fst}((1,c)) \to n_1,\quad 3 \to n_2 \\
& \nwarrow\ (1,c) \to (t'',t'''),\quad t'' \to n_1,\quad 3 \to n_2 \\
\nwarrow_{t''=1,\ t'''=c} & 1 \to n_1,\quad 3 \to n_2 \\
\nwarrow^{*}_{n_1=1,\ n_2=3} & \square
\end{array}$$

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*:

$$map \stackrel{\text{def}}{=} \lambda \underset{\tau_1\to\tau}{f}.\ \lambda \underset{\tau_1*\tau_1}{x}.\ ((\underset{\tau_1\to\tau}{f}\ \underbrace{\mathbf{fst}(\underset{\tau_1*\tau_2}{x})}_{\tau_1}), (\underset{\tau_1\to\tau}{f}\ \underbrace{\mathbf{snd}(\underset{\tau_1*\tau_2}{x})}_{\tau_2=\tau_1}))$$
$$\underbrace{\qquad\qquad}_{\tau} \qquad \underbrace{\qquad\qquad}_{\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{\tau*\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{(\tau_1*\tau_1)\to(\tau,\tau)}$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{(\tau_1\to\tau)\to(\tau_1*\tau_1)\to(\tau,\tau)}$$

We now compute the canonical form of the term $((map\ t)\ (1,2))$ where $t \stackrel{\text{def}}{=} \lambda x.\ 2\times x$:

$$\begin{aligned}
((map\ t)\ (1,2)) \to c \quad &\nwarrow (map\ t) \to \lambda y.t', \quad t'[^{(1,2)}/_y] \to c \\
&\nwarrow map \to \lambda g.t'', \quad t''[^{t}/_g] \to \lambda y.t', \quad t'[^{(1,2)}/_y] \to c \\
&\nwarrow_{g=f,\ t''=\ldots} \lambda x.\ ((t\ \mathbf{fst}(x)), (t\ \mathbf{snd}(x))) \to \lambda y.t', \\
&\qquad t'[^{(1,2)}/_y] \to c \\
&\nwarrow_{y=x,\ t'=\ldots} ((t\ \mathbf{fst}((1,2))), (t\ \mathbf{snd}((1,2)))) \to c \\
\nwarrow_{c=((t\ \mathbf{fst}((1,2))),(t\ \mathbf{snd}((1,2))))}\ &\square
\end{aligned}$$

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

$$\begin{aligned}
\text{apply}\ (f_1, d_1) &= f_1(d_1) && \text{(by definition of apply)} \\
&\sqsubseteq_E f_1(d_2) && \text{(by monotonicity of } f_1\text{)} \\
&\sqsubseteq_E f_2(d_2) && \text{(because } f_1 \sqsubseteq_{[D\to E]} f_2\text{)} \\
&= \text{apply}\ (f_2, d_2) && \text{(by definition of apply)}
\end{aligned}$$

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

$$f(d) = f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} f(d_i) = \bigsqcup_{i\in\mathbb{N}} d_i = d.$$

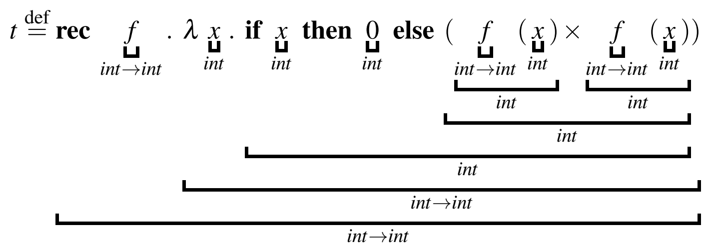
8.8 We divide the proof in two parts: first we show that $f \sqsubseteq g$ implies $f \preceq g$ and then that $f \preceq g$ implies $f \sqsubseteq g$.

For the first implication, suppose that $f \sqsubseteq g$. Given 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:



So we conclude $t : int \to int$.

The canonical form is readily obtained by unfolding once the recursive definition:

$$t \to c \qquad \nwarrow_{c = \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ (t(x) \times t(x))} \qquad \nwarrow \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ (t(x) \times t(x)) \to c \quad \square$$

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

$$
t_2 \stackrel{\text{def}}{=} \underbrace{\lambda \underbrace{x}_{\tau_1}.\ \underbrace{(\underbrace{t_1}_{\tau_1\to\tau_2}\ \underbrace{x}_{\tau_1})}_{\tau_2}}_{\tau_1\to\tau_2}
$$

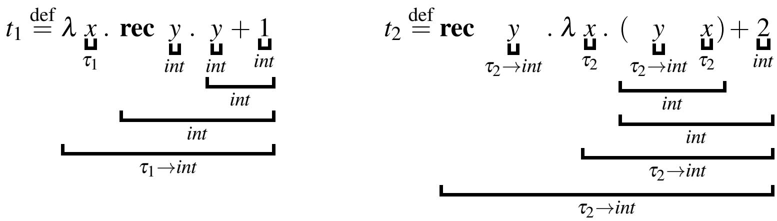
   Unless $\tau = \tau_1 \to \tau_2$ the pre-term t_2 is not typable.
2. Let us compute the denotational semantics of t_2:

$$\begin{aligned}
\llbracket t_2 \rrbracket \rho &= \left\lfloor \lambda d_x.\ \llbracket t_1\ x \rrbracket \rho[^{d_x}/_x] \right\rfloor \\
&= \left\lfloor \lambda d_x.\ \mathbf{let}\ \varphi \Leftarrow \llbracket t_1 \rrbracket \rho[^{d_x}/_x].\ \varphi(\llbracket x \rrbracket \rho[^{d_x}/_x]) \right\rfloor \\
&= \left\lfloor \lambda d_x.\ \mathbf{let}\ \varphi \Leftarrow \llbracket t_1 \rrbracket \rho[^{d_x}/_x].\ \varphi(d_x) \right\rfloor
\end{aligned}$$

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 $\llbracket t_1 \rrbracket \rho[^{d_x}/_x] = \llbracket t_1 \rrbracket \rho$.
Now, if $\llbracket t_1 \rrbracket \rho = \bot_{(V_\tau)_\bot}$, then $\llbracket t_2 \rrbracket \rho = \left\lfloor \lambda d_x.\ \bot_{(V_{\tau_2})_\bot} \right\rfloor \neq \llbracket t_1 \rrbracket \rho$.
Otherwise, it must be that $\llbracket t_1 \rrbracket \rho = \lfloor f \rfloor$ for some $f \in V_{\tau_1 \to \tau_2}$ and hence $\llbracket t_2 \rrbracket \rho = \lfloor \lambda d_x.\ f\ d_x \rfloor = \lfloor f \rfloor = \llbracket t_1 \rrbracket \rho$.

9.10

1. Let us compute the principal types for t_1 and t_2:



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:

$$\begin{aligned}
\llbracket t_1 \rrbracket \rho &= \lfloor \lambda d_x.\ \llbracket \mathbf{rec}\ y.\ y+1 \rrbracket \rho[^{d_x}/_x] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ \llbracket y+1 \rrbracket \rho[^{d_x}/_x, ^{d_y}/_y] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ \llbracket y \rrbracket \rho[^{d_x}/_x, ^{d_y}/_y] \underline{+}_\bot \llbracket 1 \rrbracket \rho[^{d_x}/_x, ^{d_y}/_y] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ d_y \underline{+}_\bot \lfloor 1 \rfloor \rfloor
\end{aligned}$$

We need to compute the fixpoint fix $\lambda d_y.\ d_y \underline{+}_\bot \lfloor 1 \rfloor$:

$$\begin{aligned}
d_0 &= \bot_{(V_{int})_\bot} \\
d_1 &= d_0 \underline{+}_\bot \lfloor 1 \rfloor = \bot_{(V_{int})_\bot} = d_0
\end{aligned}$$

From which it follows

$$\llbracket t_1 \rrbracket \rho = \lfloor \lambda d_x.\ \bot_{(V_{int})_\bot} \rfloor = \lfloor \bot_{(V_{\tau \to int})} \rfloor$$

Let us now turn our 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, recall 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

$$[\![t_1]\!]\rho = \lfloor 1 \rfloor \not\sqsubseteq \lfloor 0 \rfloor = [\![t_2]\!]\rho$$

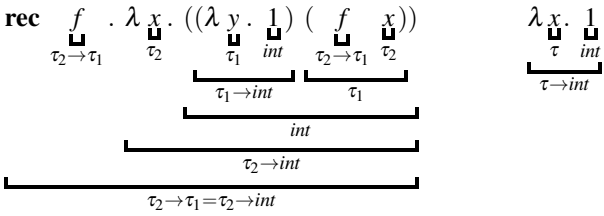
As a consequence, typable terms such as

$$t \stackrel{\text{def}}{=} \lambda x.\ \textbf{if } x \textbf{ then } 0 \textbf{ else } 1 \ :\ int \to int$$

would not be assigned a semantics in $(V_{int\to int})_\perp$ 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:



So it must be that $\tau_1 = int$ and the terms have the same type if $\tau_2 = \tau$.

The denotational semantics of t_1 requires the computation of the fixpoint:

$$\begin{aligned}
[\![t_1]\!]\rho &= \text{fix } \lambda d_f.\ [\![\lambda x.\ ((\lambda y.\ 1)\ (f\ x))]\!]\rho[^{d_f}/_f] \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ [\![((\lambda y.\ 1)\ (f\ x))]\!]\underbrace{\rho[^{d_f}/_f, ^{d_x}/_x]}_{\rho'} \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow [\![\lambda y.\ 1]\!]\rho'.\ \varphi([\![f\ x]\!]\rho')) \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow \lfloor \lambda d_y.\ \lfloor 1 \rfloor \rfloor.\ (\varphi(\textbf{let } \varphi' \Leftarrow d_f.\ \varphi'(d_x)))) \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ ((\lambda d_y.\ \lfloor 1 \rfloor)(\textbf{let } \varphi' \Leftarrow d_f.\ \varphi'(d_x))) \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor \\
f_0 &= \perp_{(V_{\tau\to int})_\perp} \\
f_1 &= \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor
\end{aligned}$$

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

$$\begin{aligned}[\![t_2]\!]\rho &= \lfloor \lambda d_x.\ [\![1]\!]\rho[{}^{d_x}/_x]\rfloor \\ &= \lfloor \lambda d_x.\ \lfloor 1\rfloor\rfloor \\ &= [\![t_1]\!]\rho\end{aligned}$$

To show that the canonical forms are different, we note that t_2 is already in canonical form, while for t_1 we have

$$t_1 \to c_1 \qquad \nwarrow_{c_1=\lambda x.\ ((\lambda y.\ 1)\ (t_1\ x))} \square \qquad \nwarrow \lambda x.\ ((\lambda y.\ 1)\ (t_1\ x)) \to c_1$$

10.2

1. We compute the denotational semantics of *map* and of $t \stackrel{\text{def}}{=} (map\ \lambda z.\ z)$:

$$\begin{aligned}[\![map]\!]\rho &= \lfloor \lambda d_f.\ [\![\lambda x.\ ((f\ \mathbf{fst}(x)),(f\ \mathbf{snd}(x)))]\!]\rho[{}^{d_f}/_f]\rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ [\![((f\ \mathbf{fst}(x)),(f\ \mathbf{snd}(x)))]\!]\underbrace{\rho[{}^{d_f}/_f,{}^{d_x}/_x]}_{\rho'}\rfloor\rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor([\![(f\ \mathbf{fst}(x))]\!]\rho',[\![(f\ \mathbf{snd}(x))]\!]\rho')\rfloor\rfloor\rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor((\mathbf{let}\ \varphi_1 \Leftarrow [\![f]\!]\rho'.\ \varphi_1([\![\mathbf{fst}(x)]\!]\rho')), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow [\![f]\!]\rho'.\ \varphi_2([\![\mathbf{snd}(x)]\!]\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 [\![x]\!]\rho'.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow d_f.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow [\![x]\!]\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 \\ [\![t]\!]\rho &= \mathbf{let}\ \varphi \Leftarrow [\![map]\!]\rho.\ \varphi([\![\lambda z.\ z]\!]\rho) \\ &= \mathbf{let}\ \varphi \Leftarrow [\![map]\!]\rho.\ \varphi(\lfloor \lambda d_z.\ [\![z]\!]\rho[{}^{d_z}/_z]\rfloor) \\ &= \mathbf{let}\ \varphi \Leftarrow [\![map]\!]\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 readily be checked that

$$[\![(t_1,t_2)]\!]\rho = \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor = [\![(t_2,t_1)]\!]\rho.$$

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:

$$\begin{array}{rl}
(t_0\ (t_1,t_2)) \to c_1 & \nwarrow\ t_0 \to \lambda x'.t,\quad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
& \nwarrow\ map \to \lambda f'.t',\quad t'[^{\lambda z.\ z}/_{f'}] \to \lambda x'.t,\quad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
& \nwarrow_{f'=f,t'=\ldots}\ \lambda x.\ (((\lambda z.\ z)\ \textbf{fst}(x)),((\lambda z.\ z)\ \textbf{snd}(x))) \to \lambda x'.t,\\
& \qquad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
& \nwarrow_{x'=x,t=\ldots}\ (((\lambda z.\ z)\ \textbf{fst}((t_1,t_2))),((\lambda z.\ z)\ \textbf{snd}((t_1,t_2)))) \to c_1\\
& \nwarrow_{c_1=(\ldots,\ldots)}\ \square\\
(t_0\ (t_2,t_1)) \to c_2 & \nwarrow^*\ (((\lambda z.\ z)\ \textbf{fst}((t_2,t_1))),((\lambda z.\ z)\ \textbf{snd}((t_2,t_1)))) \to c_2\\
& \nwarrow_{c_2=(\ldots,\ldots)}\ \square
\end{array}$$

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

$$\begin{aligned}
[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho &= \mathit{Cond}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by definition)}\\
&= \mathit{Cond}(\lfloor 0 \rfloor, [\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by inductive hypothesis)}\\
&= [\![t_0]\!]\rho && \text{(by definition of } \mathit{Cond})\\
&= [\![c_0]\!]\rho && \text{(by inductive hypothesis)}
\end{aligned}$$

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 $t \Downarrow$. However, $t \uparrow$, as

$$\begin{aligned}
t \to c &\nwarrow 0 \to 0, \quad 1 \to c, \quad \textbf{rec } x.\ x \to c'\\
&\nwarrow 1 \to c, \quad \textbf{rec } x.\ x \to c'\\
&\nwarrow_{c=1} \textbf{rec } x.\ x \to c'\\
&\nwarrow x[^{\textbf{rec } x.\ x}/_x] \to c'\\
&= \textbf{rec } x.\ x \to c'\\
&\nwarrow \cdots
\end{aligned}$$

10.13 According to the operational semantics we have

$$\begin{aligned}
\textbf{rec } x.\ t \to c &\nwarrow t[^{\textbf{rec } x.\ t}/_x] \to c\\
&= t \to c
\end{aligned}$$

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

$$\begin{aligned}
[\![\textbf{rec } x.\ t]\!]\rho &= \mathrm{fix}\ \lambda d_x.\ [\![t]\!]\rho[^{d_x}/_x]\\
&= \mathrm{fix}\ \lambda d_x.\ [\![t]\!]\rho
\end{aligned}$$

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:

$$\begin{aligned} [\![\mathbf{rec}\ x.\ t]\!]\rho &= [\![t]\!]\rho[{}^{[\![\mathbf{rec}\ x.\ t]\!]\rho}/_x] && \text{(by definition)} \\ &= [\![t[{}^{\mathbf{rec}\ x.\ t}/_x]]\!]\rho && \text{(by Substitution Lemma)} \\ &= [\![t]\!]\rho && \text{(because } x \notin \mathrm{fv}(t)\text{)} \end{aligned}$$

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 $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho \neq \bot_{\mathbb{Z}_\bot}$, there 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 the initial hypothesis). Since $c : \mathit{int}$ it must be that $c = n$ 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
$$\begin{array}{rl} (t_1\ t_0) \to c & \nwarrow\ t_1 \to \lambda x'.\ t_1',\quad t_1'[{}^{t_0}/_{x'}] \to c \\ \nwarrow_{x'=x,\ t_1'=((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))} & ((\lambda y.\ t_0)\ (\mathbf{rec}\ z.\ z)) \to c \\ & \nwarrow\ (\lambda y.\ t_0) \to \lambda y'.\ t_2,\quad t_2[{}^{(\mathbf{rec}\ z.\ z)}/_{y'}] \to c \\ \nwarrow_{y'=y,\ t_2=t_0} & t_0[{}^{(\mathbf{rec}\ z.\ z)}/_y] \to c \\ & = t_0 \to c \\ \nwarrow_{c=c_0} & \square \end{array}$$

   Whereas in the eager semantics, we have

$$
\begin{array}{l}
(t_1\ t_0) \to c \qquad\qquad \nwarrow\ t_1 \to \lambda x'.\ t_1', \quad t_0 \to c', \quad t_1'[{}^{c'}/_{x'}] \to c \\
\quad \nwarrow_{x'=x,\ t_1'=((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))}\ t_0 \to c', \quad ((\lambda y.\ c')\ (\mathbf{rec}\ z.\ z)) \to c \\
\qquad\qquad \nwarrow_{c'=c_0}\ ((\lambda y.\ c_0)\ (\mathbf{rec}\ z.\ z)) \to c \\
\qquad\qquad\qquad \nwarrow\ (\lambda y.\ c_0) \to \lambda y'.\ t_2, \quad \mathbf{rec}\ z.\ z \to c'', \quad t_2[{}^{c''}/_{y'}] \to c \\
\qquad\qquad\qquad \nwarrow_{y'=y,\ t_2=t_0}\ \mathbf{rec}\ z.\ z \to c'', \quad t_0[{}^{c''}/_{y}] \to c \\
\qquad\qquad\qquad\qquad \nwarrow\ \mathbf{rec}\ z.\ z \to c'', \quad t_0[{}^{c''}/_{y}] \to c \\
\qquad\qquad\qquad\qquad \nwarrow\ \cdots
\end{array}
$$

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 transition 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 transition 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 this value is i (in all other cases, the continuation is p_2):

$$\begin{array}{l}\overline{xr_1}.p_2 \;+\; \ldots \;+\; \overline{xr_{i-1}}.p_2 \;+\\ \overline{xr_i}.p_1 \;+\\ \overline{xr_{i+1}}.p_2 \;+\; \ldots \;+\; \overline{xr_n}.p_2\end{array}$$

Iteration: Let ϕ be the permutation that switches d and $done$. By using the recursive process, we let

$$\begin{array}{l}\mathbf{rec}\; W.\;\; \overline{xr_1}.Done \;+\; \ldots \;+\; \overline{xr_{i-1}}.Done+\\ \qquad\quad \overline{xr_i}.(p[\phi] \mid d.W)\backslash d \;+\\ \qquad\quad \overline{xr_{i+1}}.Done \;+\; \ldots \;+\; \overline{xr_n}.Done\end{array}$$

in the case the value i can be read from x, activates the continuation

$$(p[\phi] \mid d.\mathbf{rec}\; W.\;(\ldots))\backslash d$$

that executes p and (if and) when it terminates activates another instance of the recursive process. In all 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 bisimulations themselves and they are included in R.
- Take $(p_1 + p_2, q_1 + q_2) \in R$ and take μ, p such that $p_1 + p_2 \stackrel{\mu}{\rightarrow} p$. We want to prove that there exists q with $q_1 + q_2 \stackrel{\mu}{\rightarrow} q$ and $p\ R\ q$. Since $p_1 + p_2 \stackrel{\mu}{\rightarrow} p$, by the operational semantics of CCS it must be the case that either $p_1 \stackrel{\mu}{\rightarrow} p$ or $p_2 \stackrel{\mu}{\rightarrow} p$.
  - If $p_1 \stackrel{\mu}{\rightarrow} p$, since $p_1\ R_1\ q_1$, there exists q with $q_1 \stackrel{\mu}{\rightarrow} q$ and $p\ R_1\ q$. Then $q_1 + q_2 \stackrel{\mu}{\rightarrow} q$ and $(p,q) \in R_1 \subseteq R$, so we are done.
  - If $p_2 \stackrel{\mu}{\rightarrow} p$, since $p_2\ R_2\ q_2$, there exists q with $q_2 \stackrel{\mu}{\rightarrow} q$ and $p\ R_2\ q$. Then $q_1 + q_2 \stackrel{\mu}{\rightarrow} q$ 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 $q_1 + q_2 \stackrel{\mu}{\rightarrow} q$ 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 $p \stackrel{\mu}{\rightarrow} p'$. 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 are done. 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

   $$p \stackrel{\tau}{\rightarrow} p_1 \stackrel{\tau}{\rightarrow} p_2 \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow} p_n$$

   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 the 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

$$p \stackrel{\tau}{\Rightarrow} p'' \xrightarrow{\mu} p''' \stackrel{\tau}{\Rightarrow} p'$$

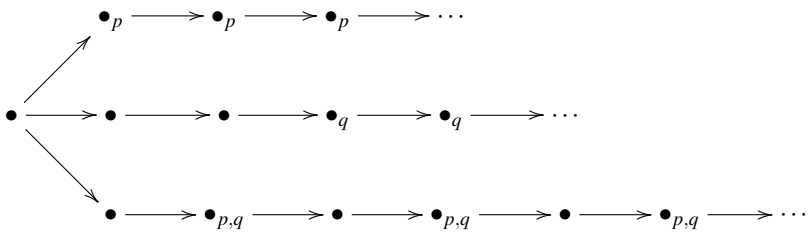
  By the argument above, we can find q'' such that $q \stackrel{\tau}{\Rightarrow} q''$ and $(p'', q'') \in R$. Since $p'' \xrightarrow{\mu} p'''$ and R is a weak bisimulation, there exists q''' such that $q'' \stackrel{\mu}{\Rightarrow} q'''$ and $(p''', q''') \in R$. By the argument above, we can find 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 subterm $\bar{x}y.p$ then $p = \mathbf{nil}$. We abbreviate $\bar{x}y.\mathbf{nil}$ as $\bar{x}y$. Let us try to encode the ordinary (synchronous) π-calculus in the 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, which have no correspondence in the 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, then we must also revise the translation of input prefixes, in order to send the acknowledgment:

$$\begin{aligned} \mathscr{A}(\overline{x}y.p) &\stackrel{\text{def}}{=} \overline{x}y \mid a(x_k).\mathscr{A}(p) \qquad \text{with } x_k \notin \text{fn}(p) \\ \mathscr{A}(x(z).q) &\stackrel{\text{def}}{=} x(z).(\overline{a}a \mid \mathscr{A}(q)) \end{aligned}$$

   Also this solution has a pitfall, because if a unique channel a is used to send all the acknowledgments, then communications can interfere with one another.
3. As a third attempt, we enforce the sharing of a private channel a for sending the acknowledgment. The sender creates the channel and transmits 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

$$\pi \ ::= \ \tau \mid x(z_1,...,z_n) \mid \overline{x}\langle y_1,...,y_n\rangle$$

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 ourselves to encoding only such well-typed processes. Let us try to encode the polyadic π-calculus in the 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:

$$P \ ::= \ \mathbf{nil} \mid \pi.P \mid P|Q \mid (y)P \mid Y$$
$$\pi \ ::= \ \tau \mid x(y) \mid \overline{x}y \mid x(Y) \mid \overline{x}\langle P\rangle$$

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 assign it to the process variable Y. Without delving into the details of the operational and abstract semantics for HOπ, higher-order communication can be realised by transitions such as:

$$\overline{x}\langle P\rangle.Q \mid x(Y).R \xrightarrow{\tau} Q \mid R[{}^{P}/_{Y}]$$

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 to avoid interference from the environment.

To encode HOπ in the (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 us to encode process input prefixes such as $x(Y)$ 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

$$\begin{aligned}
\mathscr{H}(\overline{x}\langle P\rangle.Q) &\stackrel{\text{def}}{=} (p)(\ \mathscr{H}(Q) \mid !p(x_p).\mathscr{H}(P)\ ) \qquad \text{with } p, x_p \notin \mathrm{fn}(P)\\
\mathscr{H}(x(Y).R) &\stackrel{\text{def}}{=} x(x_Y).\mathscr{H}(R) \qquad \text{with } x_Y \notin \mathrm{fn}(R)\\
\mathscr{H}(Y) &\stackrel{\text{def}}{=} \overline{x_Y}x_Y.\mathbf{nil}
\end{aligned}$$

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 $x \notin \mathrm{fn}(p)$)

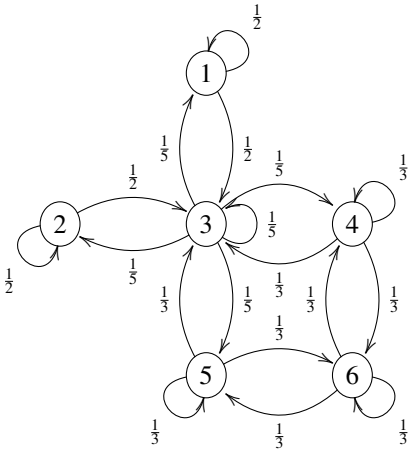
$$(x)p \equiv (x)(p \mid \mathbf{nil}) \equiv p \mid (x)\mathbf{nil} \equiv p \mid \mathbf{nil} \equiv p$$

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 $\mathrm{fn}(p) = \varnothing$ and $\mathrm{fn}(q) = \{y\}$, 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: $\pi_3 = \frac{5}{18}$, $\pi_1 = \pi_2 = \frac{1}{9}$ and $\pi_4 = \pi_5 = \pi_6 = \frac{1}{6}$, i.e., $\pi = \left|\frac{1}{9}\ \frac{1}{9}\ \frac{5}{18}\ \frac{1}{6}\ \frac{1}{6}\ \frac{1}{6}\right|$.

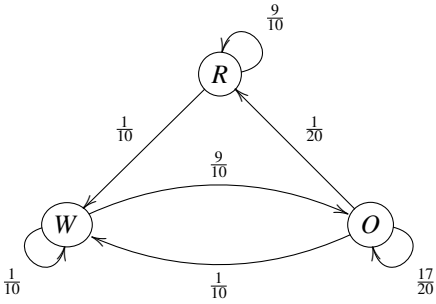
3. We have

$$\pi^{(0)} = \left|1\ 0\ 0\ 0\ 0\ 0\right|$$
$$\pi^{(1)} = \pi^{(0)}P = \left|\tfrac{1}{2}\ 0\ \tfrac{1}{2}\ 0\ 0\ 0\right|$$
$$\pi^{(2)} = \pi^{(1)}P = \left|\tfrac{1}{4}+\tfrac{1}{10}\ \tfrac{1}{10}\ \tfrac{1}{4}+\tfrac{1}{10}\ \tfrac{1}{10}\ \tfrac{1}{10}\ 0\right| = \left|\tfrac{7}{20}\ \tfrac{1}{10}\ \tfrac{7}{20}\ \tfrac{1}{10}\ \tfrac{1}{10}\ 0\right|$$
$$\pi^{(3)} = \pi^{(2)}P = \left|\ldots\ \tfrac{1}{30}+\tfrac{1}{30}\right| = \left|\ldots\ \tfrac{1}{15}\right|$$

Thus, the probability of finding 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 $\pi^{(t)} = \left|0\ 1\ 0\right|$. Then, $\pi^{(t+1)} = \pi^{(t)}P = \left|0\ \frac{1}{10}\ \frac{9}{10}\right|$ and the probability of being 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 of being operating in 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 $\pi^{(0)} = \left|p\ q\right|$ for some $p, q \in [0,1]$ with $q = 1-p$. We have $\pi^{(1)} = \pi^{(0)}P = \left|q\ p\right|$ and $\pi^{(2)} = \pi^{(1)}P = \left|p\ q\right| = \pi^{(0)}$. 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 on 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 that $d(\ell, s') = 0$;
  - 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 that $d_\ell(\ell', s') = 0$ 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 that $d_{(\ell,d)}(\ell', s') = 0$ 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 assign 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

$$\begin{array}{rlrl}
\gamma(s_0)(2€)(I_2) &= \frac{2}{3} & \gamma(s_2)(2€)(I_2) &= \frac{2}{3} \\
\gamma(s_0)(2€)(I_3) &= \frac{1}{3} & \gamma(s_2)(2€)(I_3) &= \frac{1}{3} \\
\gamma(s_0)(3.5€)(I_2) &= \frac{1}{3} & \gamma(s_2)(3.5€)(I_2) &= \frac{1}{3} \\
\gamma(s_0)(3.5€)(I_3) &= \frac{2}{3} & \gamma(s_2)(3.5€)(I_3) &= \frac{1}{3} + \frac{1}{3} = \frac{2}{3} \\
\gamma(s_0')(coffee)(I_1) &= 1 & \gamma(s_2')(coffee)(I_1) &= 1 \\
\gamma(s_0'')(beer)(I_1) &= 1 & \gamma(s_2'')(beer)(I_1) &= 1 = \gamma(s_2''')(beer)(I_1)
\end{array}$$

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: