Roberto Bruni, Ugo Montanari

# Models of Computation

– Monograph –

March 1, 2016

Springer

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

*Alan Turing*[1]

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

# Contents

# Acronyms

|  |  |
|---|---|
| $\sim$ | operational equivalence in IMP (see Definition 3.3) |
| $\equiv_{den}$ | denotational equivalence in HOFL (see Definition 10.4) |
| $\equiv_{op}$ | operational equivalence in HOFL (see Definition 10.3) |
| $\simeq$ | CCS strong bisimilarity (see Definition 11.5) |
| $\approx$ | CCS weak bisimilarity (see Definition 11.16) |
| $\cong$ | CCS weak observational congruence (see Section 11.7.2) |
| $\approx_d$ | CCS dynamic bisimilarity (see Definition 11.17) |
| $\overset{\circ}{\sim}_E$ | $\pi$-calculus early bisimilarity (see Definition 13.3) |
| $\overset{\circ}{\sim}_L$ | $\pi$-calculus late bisimilarity (see Definition 13.4) |
| $\sim_E$ | $\pi$-calculus strong early full bisimilarity (see Section 13.5.3) |
| $\sim_L$ | $\pi$-calculus strong late full bisimilarity (see Section 13.5.3) |
| $\overset{\bullet}{\approx}_E$ | $\pi$-calculus weak early bisimilarity (see Section 13.5.4) |
| $\overset{\bullet}{\approx}_L$ | $\pi$-calculus weak late bisimilarity (see Section 13.5.4) |
| $\mathscr{A}$ | interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1) |
| *ack* | Ackermann function (see Example 4.18) |
| *Aexp* | set of IMP arithmetic expressions (see Chapter 3) |
| $\mathscr{B}$ | interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2) |
| *Bexp* | set of IMP boolean expressions (see Chapter 3) |
| $\mathbb{B}$ | set of booleans |
| $\mathscr{C}$ | interpretation function for the denotational semantics of IMP commands (see Section 6.2.3) |
| CCS | Calculus of Communicating Systems (see Chapter 11) |
| *Com* | set of IMP commands (see Chapter 3) |
| CPO | Complete Partial Order (see Definition 5.11) |
| $\mathrm{CPO}_\perp$ | Complete Partial Order with bottom (see Definition 5.12) |
| CSP | Communicating Sequential Processes (see Section 16.2) |
| CTL | Computation Tree Logic (see Section 12.1.2) |
| CTMC | Continuous Time Markov Chain (see Definition 14.15) |

| | |
|---|---|
| DTMC | Discrete Time Markov Chain (see Definition **??**) |
| *Env* | set of HOFL environments (see Chapter 9) |
| fix | (least) fixpoint (see Definition 5.2.2) |
| FIX | (greatest) fixpoint |
| gcd | greatest common divisor |
| HML | Hennessy-Milner modal Logic (see Section 11.5) |
| HM-Logic | Hennessy-Milner modal Logic (see Section 11.5) |
| HOFL | A Higher-Order Functional Language (see Chapter 7) |
| IMP | A simple IMPerative language (see Chapter 3) |
| *int* | integer type in HOFL (see Definition 7.2) |
| **Loc** | set of locations (see Chapter 3) |
| LTL | Linear Temporal Logic (see Section 12.1.1) |
| LTS | Labelled Transition System (see Definition 11.2) |
| lub | least upper bound (see Definition 5.7) |
| $\mathbb{N}$ | set of natural numbers |
| $\mathscr{P}$ | set of closed CCS processes (see Definition 11.1) |
| PEPA | Performance Evaluation Process Algebra (see Chapter 16) |
| **Pf** | set of partial functions on natural numbers (see Example 5.10) |
| **PI** | set of partial injective functions on natural numbers (see Problem 5.11) |
| PO | Partial Order (see Definition 5.1) |
| PTS | Probabilistic Transition System (see Section 14.3.2) |
| $\mathbb{R}$ | set of real numbers |
| $\mathscr{T}$ | set of HOFL types (see Definition 7.2) |
| **Tf** | set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.11) |
| *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 exposes some important subjects covered in the book, like 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.* For example, the alphabet of roman numerals, the numeric system used in ancient Rome, consists of seven letters from the Latin alphabet. A value is assigned to each letter (see Table 1.1) and a number $n$ is expressed by juxtaposing the letters such that the sum of their values is $n$. Not all sequences are valid. 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, like IIII, subtractive notation is used: by placing a symbol with smaller value $u$ to the left of a symbol with higher value $v$ we indicate 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. Thus IX and XI are both correct sequences, with values 9 and 11, respectively, while 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 a roman numeral to a positive natural number give the semantics of roman numerals.

Table 1.1: Alphabet of roman numerals

| Symbol | I | V | X | L | C | D | M |
|---|---|---|---|---|---|---|---|
| Value | 1 | 5 | 10 | 50 | 100 | 500 | 1000 |

### 1.1.1 Syntax and Types

The syntax is concerned with the alphabet of symbols, and with the grammatical structure of programs. 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 studied in every computer science degree and are now widely applied tools for defining the syntax of formal languages. Such a formalisation can be found in the appendix of most language manuals.

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. Different types can be defined over the same language. Types are often presented as set of logic rules, called *type systems* that are used to assign a type unambiguously to each program and computed value.

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, which auxiliary tools are available (syntax checkers, debuggers, etc.). Unfortunately this leaves space to different interpretations that can ultimately lead to discordant implementations of the same language or to compilers that rely on wrong code optimisation strategies.

If a language's semantics would be formalised as well, it could appear in the language manual, too and solve any ambiguity. 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 parsers for grammars.

### 1.1.2 Semantics

It seems the word 'semantics' was introduced in 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 language.

In Computer Science, the semantics is concerned with

the study of the meaning of (well-typed) programs.

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

Since programmers can write programs that “work as expected”, once they have been thoroughfully tested,

what do we gain by formalising the semantics of a programming language?

An easy answer is that today, in the era of the Internet of Things, our lives, the machines and devices we use, and the entire world run on software: It is not enough to require that medical implants, 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 users, as a guide to write syntactically correct programs, and
   - to implementors, who must write a correct parser for the language’s compiler.
2. it permits a formal analysis of many properties, like finding and resolving parsing ambiguities.
3. it can be used as input to a compiler front-end generating tool, such as Yacc, Bison, Xtext. In this way, from the syntax definition one can automatically derive an implementation of the front-end of the language’s compiler.

Providing a formal semantics definition of a programming language can then lead to similar benefits:

1. it standardises a machine-independent specification of the language; this is crucial:
   - to users, for understanding and improving the programs they write, and
   - to implementors, who must write a correct and efficient code generator for the language’s compiler.
2. it permits a formal analysis of program properties, such as type preservation, termination, specification compliance or program equivalence.
3. it can be used as input to a compiler back-end generating tool. In this way, the semantics definition gives also the (prototypal and possibly inefficient) implementation of the back end of the language’s compiler.

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 a specific piece of commercial hardware, 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, 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?

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 one can prove such important properties like termination of program execution, determinism of the computed result, program equivalence, or program correctness.

*Example 1.2.* We can hardly claim to know that two programs mean the same 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 viceversa) in a program, without changing its meaning? Under which circumstances?[1]

## *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 simply the activity of trying to define the meaning of program constructions precisely can reveal all kinds of subtleties of which it is important to be aware.

Unlike the acceptance of BNF as a standard definition method for syntax, it appears unlikely 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.

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

Different semantic styles and models have been developed for different purposes, depending on the main task to be addressed. 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 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.

### *1.1.4 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).[3] A later example was the definition of the semantics of Algol 68 in terms of a hypothetical computer over which program actions are executed.

In 1981, Gordon Plotkin (1946–) introduced the structural operational semantics style (SOS-style) in a technical report,[4] which by now is one of the most cited technical reports in computer science, only recently revised and reprinted in a journal.[5]

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.

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 could in principle be interpreted by a logic programming system, such as Prolog.[6]

Because of the strong connection with the syntactic structure and the fact that the mathematics involved is usually not very complicated, operational semantics can lead

[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] McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM 3(4):184-195, 1960.

[4] Plotkin, G.D.: A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[5] Plotkin, G.D.: A Structural Approach to Operational Semantics. J. Log. Algebr. Program. 60–61:17–139, 2004.

[6] Apart from issues about performance and the fact that Prolog is not complete, because it exploits a depth-first exploration strategy: backtracking out of wrong attempted transition sequences is only possible if they are finite.

to descriptions that are understandable even for non-specialists: SOS-style operational semantics can provide the programmer with a concise and accurate description of what the language constructs do, because it is syntax-oriented, inductive and easy to grasp.

### 1.1.5 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[7] of Christopher Strachey (1916–1975) and Dana Scott (1932–) in the late 1960s and in fact it is sometimes called Scott-Strachey semantics.

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.

### 1.1.6 Axiomatic Semantics

Instead of directly assigning a meaning to a program, axiomatic semantics gives a description of the constructs in a programming language by providing axioms that are satisfied by these constructs. Axiomatic semantics poses 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 Tony Hoare (1934–) on structured imper-

[7] Scott, D.S.: Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, Oxford, England, November 1970. Scott D.S., Strachey, C.: Toward a mathematical semantics for computer languages. Oxford Programming Research Group Technical Monograph. PRG-6. 1971.

[8] Floyd, R.W.: Assigning meanings to programs. Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 19–31, 1967.

ative programs[9], and in fact it is sometimes referred to as Floyd-Hoare logic. The basic idea is that programs and 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 preconditions are 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:

| | |
|---|---|
| Partial correctness: | 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.) |
| Total correctness: | 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. |

One 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 Semantics Methods: Numerical Expressions

We can give a first, informal overview of the different flavours of semantics styles we will consider in this book by taking a simple example of numerical expressions.[10] 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:

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

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. To differentiate between numerals (like $\mathtt{5}$) and numbers (like 5) we use different fonts.

*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,

[9] Hoare, C. A. R.: An axiomatic basis for computer programming. Communications of the ACM 12 (10): 576–580,1969.

[10] The example has been inspired from some course notes on the “Semantics of programming languages”, by Matthew Hennessy.

- the string “101” in binary notation,
- the string “V” in roman numerals.

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

- any numeral `n` is also an expression;
- if we have already constructed two expressions $e_0$ and $e_1$, then $e_0 \oplus e_1$ is also an expression;
- if we have already constructed two expressions $e_0$ and $e_1$, then $e_0 \otimes e_1$ is also an expression.

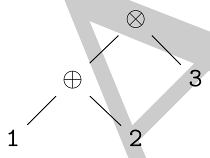
In the book we will always use abstract syntax representations, as if all concrete terms were parsed before we start to work with them.

*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 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

$$\mathtt{1} \oplus \mathtt{2} \otimes \mathtt{3}$$

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 $(\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{3}$ to save space while avoiding ambiguities.

### 1.2.1 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 `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.[11]

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

- $\mathtt{2}$ is evaluated to 2
- $(\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{3}$ is evaluated to 9
- $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$ is evaluated to 21

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

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

### *1.2.2 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. Small-step semantics provides an abstraction of how the program is executed on a machine: we ignore details like the use of registers and addresses for variables. 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* hold true then the *conclusion* can be drawn, where the premises consist of one, none or more judgements and the condition is a single judgement. The *rule-name* is just a convenient label that can be used to refer 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

[11] 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

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

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

as witnessed by the formal derivations

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

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 so to guarantee, e.g., that the left-most 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 left-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:

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

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 exists $e_1, e_2, \ldots, e_{k-1}$ such that we can derive the judgements

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

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

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

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

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 $\texttt{n}$, we have $\texttt{n} \not\to$, as no conclusion of the inference rules has a numeral as 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

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

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

In our example we can derive the judgement

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

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

### *1.2.3 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 in natural semantics is 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

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

Fig. 1.2: Natural semantics for numerical expressions

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 often directly correspond to an efficient implementation of an interpreter for the language. In our running example, we might want to disregard the individual steps that led to the result and be interested only in the final outcome, i.e., in the predicate $E \to^* \mathtt{n}$.

Using natural semantics we can formalise derivations of such longer computations by exploiting inference rules. To avoid confusion, here we denote the judgements using the notation

$$e \downarrow \mathtt{n}$$

to be read as: *the expression e is (eventually) evaluated to* 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

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

can be derived as follows:

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

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.

### *1.2.4 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 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 syntax category).

In our running example, we can define two semantics functions:

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

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!

*Remark 1.3.* When we will study more complex languages, we will find that we need 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 mapped input into output. To give denotations to recursively defined programs, Scott proposed working with continuous functions between domains, specifically complete partial orders.

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 the 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.

### 1.2.5 Semantic Equivalence

We have now available three different semantics for numerical expressions:

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

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 $e \downarrow \mathtt{n}$, while 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 $E \downarrow \mathtt{n}$, while is trivially satisfied by $\mathscr{E}[\![e]\!]$.
3. Is it true that $e \to^* \mathtt{n}$ iff $e \downarrow \mathtt{n}$?
   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 $\mathtt{n}$, $e_0 \downarrow \mathtt{n}$ iff $e_1 \downarrow \mathtt{n}$.
- 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.

We can also exploit the formal definitions to prove or disprove that two expressions are equivalent.

### 1.2.6 Expressions with Variables

Suppose now we want to extend numerical expressions with the possibility to include formal paramters in them, drawn from an infinite set $X$.

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

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

Operationally, we must provide such an information to the machine, e.g., in 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 $\sigma$. The corresponding small-/big-step rules for variable would then look like:

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

**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:

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

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

**Exercise 1.2.** The reader is invited to restate the consistency between the various semantics and the operational notions of equivalences between expressions by 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 for 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 explain some possible benefits for each of the above categories.

### *1.3.1 Language Design*

The effort spent in 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 presenting each feature can be discovered.

The worst form of design errors are unintentional ones, where the language behaves in a way that is not expected and even less desired by its designers (for example, a supposedly safe static type-checking mechanism that is not such).

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 instruments for detecting problems in language design.

### *1.3.2 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, like the correctness of compilers optimisations.

A common phenomenon is the presence of underspecified behaviour in certain circumstances. In practice, such underspecified behaviours can mine programs portability from one implementation 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 itself 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.

### 1.3.3 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 mean 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 functions

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


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

### 1.3.4 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 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, is sustained by a rich mathematical theory.
- Axiomatic semantics can lead to strikingly elegant proof systems, useful in developing as well as verifying programs.[12]

[12] 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.

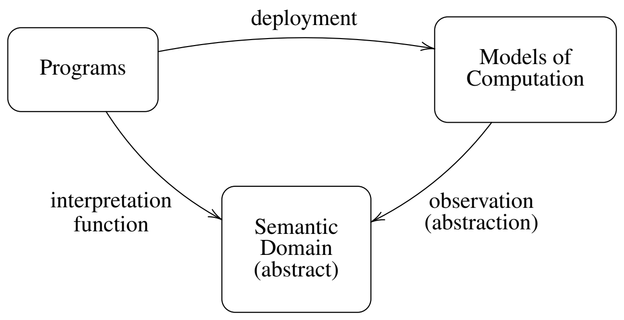


Fig. 1.3: Programs, models and domains

A longstanding 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.

In the book we show how to relate different kinds of semantics and program equivalences, reconciling whenever possible the operational, denotational and logic views.

## 1.4 Content Overview

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 on program properties.

In this book we focus on the operational and denotational semantics, present the fundamental ideas and methods behind these approaches and stress their relationship, by formulating and proving some relevant correspondence theorems. Figure 1.3 sketches the general scenario, where we have programs that can be deployed / executed over suitable operational models and that can be assigned some abstract semantics (e.g., interpreted in some mathematical domain). The scenario is possibly completed by the definition of a suitable abstraction of the operational models (of deployed programs) that can be related to the direct interpretation of programs (e.g., showing that the induced notion of program equivalence is the same in both cases).

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 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. We call an interpretation function the mapping from program syntax to program semantics and we address questions that arise naturally, like establishing if two programs are equivalent, i.e., if they have the same semantics or not. Like a numerical expression can be evaluated to a value, the interpretation function for a typical imperative language is a mapping that, given a program, returns a function from its initial states to its final states. Here the situation is a bit more complicated by the fact that 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.

If we want to prove non-trivial properties of a program or of a class of programs, we usually have to use *induction* mechanisms which allow us to prove properties of elements of an infinite set (like the steps of a run, or the programs in a class). The most general notion of induction is the so called *well-founded induction* (or *Noether* induction) and we derive from it all the other inductions principles.

Defining a program by *structural recursion* means to specify 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.

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:

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

We are looking for the meanings of $p$ which satisfy this equation, which in general can have one, none or multiple solutions. In order to solve this problem and guarantee existence and uniqueness of a best solution, we resort to the *fixpoint* theory of *complete partial orders with bottom* and of *continuous* functions.

We use these two paradigms for: 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 define their operational semantics, their denotational semantics and finally, to some extent, we prove the correspondence between the two. The fixpoint theory for HOFL is more complex because we are working on a more general situation where functions are first class citizens.

The models we use for IMP and HOFL are not appropriate for concurrent and interactive systems, like the very common network based applications: on the one hand we want their behaviour not to depend as much as possible on the speed of processes, on the other hand we want to permit infinite computations. So we do not consider time explicitly, but we have to introduce nondeterminism to account for races between concurrent processes.

The typical models for nondeterminism and infinite computations are *(labelled) transition systems*.

$$p \xrightarrow{\mu} q$$

In the figure above, we have a transition (system) with two states $p$ and $q$ and a transition from $p$ to $q$ labelled with a suitable action $\mu$. However, from the outside we can just observe the action $\mu$ associated to the transition and not the identity of states. Equivalent programs are represented by (initial) states which have correspondent observable transitions. The language that we employ in this setting is a *process algebra* called CCS (*Calculus for Communicating Systems*), and its most used notion of observational equivalence is called *bisimilarity*. Interestingly, we can draw some analogies between the fixpoint theory and bisimilarity. We investigate also *temporal* and *modal logics* designed to conveniently express properties of such systems. For example, we show that bisimilarity can be characterised in terms of a modal logic, called Hennessy-Milner logic.

Then, we study systems whose communication structure can change during execution. These systems are called *open-ended*. As our case study, we present the $\pi$-calculus, which extends CCS. The $\pi$-calculus is quite expressive, due to its ability to create and to transmit new names, which can represent ports, links, and also session names, passwords and so on in security applications.

Finally, in the last part of the book we focus on probabilistic models, where we trade nondeterminism for probability distributions, which we associate to choice points. We also present stochastic models, where actions take place in a continuous time setting, with an exponential distribution. We extend the notion of bisimilarity to handle these systems and extend Hennessy-Milner logic to Larsen-Skou logic. To specify systems in a compositional way, we link process algebras to probability by presenting a tool supported formalism called PEPA. Probabilistic/stochastic models find applications in many fields, e.g., in performance evaluation, decision support systems and system biology.

### *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 disproving it, we can again exhibit a single counterexample).

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.

In such 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 sub-terms, for any smaller derivation, for any computation whose length is smaller or equal than $n$). |

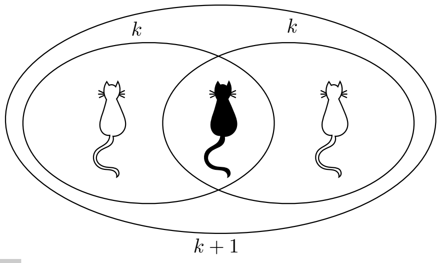
**Exercise 1.3.** Induction can be deceptive. Let us consider the following argument for proving that all cats are the same colour.

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 of the line. By inductive hypothesis, the cats in the two groups are the same colours. 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 $n \in \omega$.

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, or in the special case of rule induction.

| | |
|---|---|
| base cases: | $P$ holds for the conclusion of each axioms, 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 program by *structural recursion* means to specify 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 like:

$$[\![p]\!] = f([\![p]\!]). \tag{1.2}$$

In general, it can happen than such equations have none, 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 $M \supseteq \{1\}$ |

In order to provide a general solution to this kind of problems, 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, a special 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 over 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

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

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 $\perp$, which is less defined than any other element: in our example about partial function, 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-behaving 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.

The 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.

### 1.4.3 Bisimulation

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 best suited as a criterion for program equivalence, because it does not account properly for non-terminating computations and non-deterministic behaviour. To see this, consider the two labelled transition systems below, which can be thought to model the behaviour of two different coffee machines:

$$\begin{tikzcd}
p_0 \arrow[d, "\text{coin}"] & & & q_0 \arrow[dl, bend right, "\text{coin}"'] \arrow[dr, bend left, "\text{coin}"] & \\
p_1 \arrow[u, bend left=60, "\text{coffee}"] \arrow[u, bend right=60, "\text{tea}"'] & & q_1 \arrow[ur, bend left=60, "\text{coffee}"] & & q_2 \arrow[ul, bend right=60, "\text{tea}"']
\end{tikzcd}$$

It is evident that any sequence of actions that is executable by the first machine can be also executed on 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 right leaves no choice to the user.

We show that a suitable notion of equivalence between concurrent, interactive systems can be defined as an *observational equivalence* called *bisimulation*: it takes into account the branching structure of labelled transition systems as well as infinite computations. Interestingly, there is a nice connection between fixpoint theory and the definition of the coarsest bisimulation equivalence, called *bisimilarity*. Moreover, bisimilarity finds a logical counterparts 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.

### 1.4.4 Temporal and Modal Logics

Modal logics were conceived in Philosophy to study different *modes of truth*, like an assertion being false in the current world but *possibly* true in some alternate world, or another to *always* hold true in all worlds. Temporal logics are an instance of modal logics to reason 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 is *always* true, in all the 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 *$\mu$-calculus*, that encompasses LTL and CTL. The definition of the semantics of $\mu$-calculus exploits again the principles of domain theory and fixpoint computation.

### *1.4.5 Probabilistic Systems*

Probability theory is playing 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 non-deterministic 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 non-determinism and we present several ways in which this can be achieved.

A compelling case of probabilistic systems is given by *Markov chains*, which represents random processes over time. We study two kinds of Markov chains, which differ for the way in which time is represented (discrete vs continuous) and focus on homogeneous chains only, where the distribution depends on the current state of the system, but not by the current time. For example, in some special cases, Markov chain can be used to estimate the probability to find the system in a given state on the long run or the probability that the system will not its change state in some time.

By analogy with labelled transition systems we are also able to define suitable notions of bisimulation and the analogous 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, which can be used to define stochastic systems in a structured and compositional way as well as by refinement from specifications.

## 1.5 Chapters 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 models non-deterministic ones. The difference will emerge clear during the reading.

- Computational models for imperative languages, exemplified over 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 declined to other widely used induction principles, like weak and strong mathematical induction, structural induction and rule induction.
  - In Chapter 5 the mathematical basis for denotational semantics are presented, including the concepts and properties of complete partial orders, of least upper bounds, and of monotone and continuous functions. In particular this chapter contains Kleene fixpoint theorem that is used extensively in the rest of the monograph and the definition of the immediate consequence operator associated with a logical system and exploited in Chapter 6.
  - In Chapter 6 the foundations introduced in Chapter 5 are exploited to define the denotational semantics of IMP and 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 of their two induced equivalences over processes. The chapter is concluded by presenting Scott principle of *computational induction* for proving *inclusive* properties.
- Computational models for functional languages, exemplified over HOFL
  - In Chapter 7 we shift from the imperative style of programming to the declarative one. After presenting the $\lambda$-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 abstraction. 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 argument 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 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 / non-deterministic / interactive languages, exemplified over CCS and pi-calculus
  - In Chapter 11 we shift the focus from sequential systems to concurrent and interactive ones. The process algebra CCS is introduced which allows to describe concurrent communicating systems. Such systems communicates by message passing over named channels. Their operational semantics is defined in the small-step style, because infinite computations must be accounted for. Each communicating process is assigned a labelled transition system by inference rules in the SOS-style and several equivalences over such transition systems are discussed. In particular the notion of observational 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 expressing properties about finite and infinite computations. First the temporal logics LTL and CTL are presented, and then the more expressive $\mu$-calculus is studied. The notion of satisfaction for $\mu$-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 $\pi$-calculus, define its small-step operational semantics and introduce several notions of bisimulation equivalence.
- Computational models for probabilistic and stochastic process calculi
  - In Chapter 14 we shift the focus from non-deterministic systems to probabilistic ones. After introducing the basics of measure theory and the notions of random process and Markow property, two classes of random processes are studied, which differ for the way in which time is represented: DTMC (discrete time) and CTMC (continuous time). In both cases, it is studied how to compute stationary probability distribution over the possible states and suitable notion of bisimulation equivalence.
  - In Chapter 15, the various possibilities for defining probabilistic models of computation with observable actions and sources of non-determinism 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 to each PEPA process.