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.

© Springer International Publishing Switzerland 2017

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

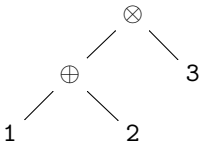
[4] The example has been inspired by some course notes on the "Semantics of programming languages", by Matthew Hennessy.

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).

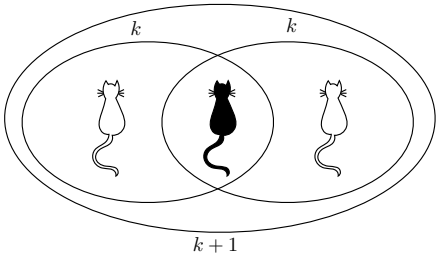
Exercise 1.3. Induction can be deceptive. Let us consider the following argument for proving that all cats are the same colour.

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.

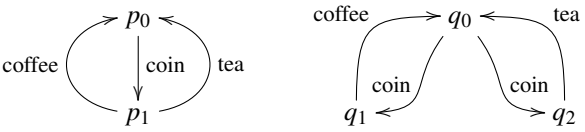
Kleene's fixpoint theorem ensures that when continuous functions are considered over complete partial orders (with bottom) then a suitable *least* fixpoint exists, and tells us how to compute it. The fixpoint theory is first applied to the case of IMP semantics and then extended to handle HOFL. The case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

http://www.springer.com/978-3-319-42898-7

Models of Computation

Bruni, R.; Montanari, U.

2017, XXII, 395 p. 34 illus., 1 illus. in color., Hardcover
ISBN: 978-3-319-42898-7