Chapter 1
Introduction

Abstract This introductory chapter outlines the main motivations for the study of concurrency theory and the differences with respect to the theory of sequential computation. It also reports the structure of the book and how to use it. Finally, some background material is briefly surveyed.

1.1 Motivation

Computer systems, implemented in hardware or software or as a combination of both, are supposed to offer certain well-specified services, so that their users can safely rely on them. However, often a computer system is not equipped with a proof that the specified service or property is guaranteed. In order to do so, one has first to define an abstract semantic model of the system (the *specification*), that can be used to study whether it satisfies the requested property. If so, then one has to use such a specification as the reference model to build the actual executable *implementation*, and possibly prove that the implementation is compliant with the specification. We call this production methodology the *specification-verification-implementation* methodology.

This kind of production methodology is largely used in more traditional and well-established engineering disciplines, such as in construction engineering, where a model of a construction, e.g., a bridge, is always designed, studied and proved correct, before being constructed. By contrast, in computer science and engineering this approach has been used extensively only recently, after some astonishing incidents in the 1990s, such as Intel's Pentium II bug in the floating point division unit in 1994. It is still common practice today to go directly to implementation: too often the specification-verification phases are missing (or are only very sketchy and informal), and correctness of the implementation is checked by testing a posteriori; however, as Dijkstra [Dij69] observed: "testing can be used to show the presence of bugs, but never to show their absence". Therefore, if the formal guarantee of correctness is a necessary requirement of the system, a formal specification must be

provided and used as a basis to prove the correctness of the design first, and then, possibly, also of the implementation.

An important reason why the specification-verification-implementation methodology is not so widespread in computer engineering is the current limitations of the theoretical tools that can be used in support. On the one hand, semantic theories for modeling computer systems are often not easy, or are even mathematically difficult, so that an engineer would certainly not spend time on it, unless the payoff is very rewarding. In some cases, indeed, the effort is worthwhile: nowadays there is an increasing number of success stories, mainly related to hardware verification. On the other hand, there are intrinsic mathematical limitations to verification that are rooted in classic undecidable problems of computability theory, such as the *halting problem* (see Section 1.3.5). Therefore, in some cases, we are forced to live with partially unverified systems.

This book aims at offering a simple, introductory theory of concurrent, reactive systems that is mathematically well-defined, rich enough to offer mathematical tools for verification and expressive enough to model nontrivial, sometimes even complex systems. It is based on the semantic model of *labeled transition systems* [Kel76] and on the language CCS, proposed by Robin Milner [Mil80, Mil89, Mil99]. The main verification technique is based on *equivalence-checking*, where an abstract model of a system, described as a CCS process, is compared with a more detailed implementation of it, expressed in the same language. We will see that this technique is useful in some remarkable cases.

Of course, this simple theory does not cover all the possible aspects of the behavior of real-life systems; for instance, we are not dealing with real-time or mobility issues; nonetheless, extensions of this theory to include such additional features are possible, already well-investigated and can be profitably studied in more specific books, such as the second part of [AILS07] for real time or [SW01] for mobility.

The following subsections provide a historical perspective on the problem of the semantics of concurrency, which has led to the ideas that are at the base of the theory presented in this textbook.

1.1.1 Sequentiality, Nondeterminism and Concurrency

Classical programming languages, such as Pascal [JW+91], are sometimes denoted as *sequential*, to express their distinguished feature that any of their programs runs in isolation, without any interference by other programs that can run concurrently. Non-termination is considered a bad feature of such programs, as the goal of a sequential program is to compute a result; moreover, in case of termination, the result is unique because the computation is *deterministic*: at any time instant, the next computation step is uniquely determined. Therefore, the semantics of a sequential program is rather intuitive: it is roughly a (partial) function from the input values (or initial values of the program variables) to the output values (or final values of the program variables), if any; the operational behavior of a sequential program

(i.e., *how* it computes) can be safely abstracted to a function (*what* it computes), with no details about the intermediate states of the computation. Hence, functions are the correct *semantic model* for sequential programs and the motto *programs-as-functions* well characterizes sequential programming.

Two sequential programs are *equivalent* if their semantics is the exact same function, independently of their operational behavior. For instance, let **:=** denote the assignment operator and **;** the sequential composition operator; then, the following two program fragments are equivalent

$$x := 1 \qquad \text{we call } p, \text{ and} \qquad x := 0 \ ; \ x := x+1 \qquad \text{we call } q$$

as they both compute the same function f: whatever initial value is attributed to x, at the end of the computation x holds value 1. A bit more formally, function f is a function that maps an association of the form (x,n), where x is the unique program variable and n is its initial value, to an association $(x,1)$, as 1 is its final value. More generally, a *store* s is a function from program variables to values, and the semantics of a sequential program is a function from stores (specifying the initial values of the program variables) to stores (specifying the final values of the variables). Therefore, formally, function f can be defined as $f(s) = s[1/x]$, meaning that given any initial store s, the final store is s where the association for x is updated to $(x,1)$.

This semantic equivalence is also a *congruence*, i.e., it is preserved by the operator of the language. For instance, since p and q above are equivalent, we have that $p;p$ and $p;q$ are equivalent (and they are both equivalent to p); in general, for any program r, we are sure that both $r;p$ and $r;q$ are equivalent, as well as $p;r$ and $q;r$.

The semantics of sequential programs is defined in a *compositional* way, meaning that for each *syntactic operator* of the language there is a corresponding *semantic operator* over functions. For instance, consider the compound program $r;t$, where the execution of program r is followed by the execution of program t, according to the syntactic operator of sequential composition. Then, if we assume that the semantics of r is function f_r and the semantics of t is function f_t, then the semantics of $r;t$ is obtained by combining f_r and f_t by means of the semantic operator of functional composition: $f_r \circ f_t$. For instance, considering the program fragments p and q above, the semantics of $p;q$ is function $f \circ f = f$, where f is the function $f(s) = s[1/x]$.

By extending a sequential programming language with an operator of parallel composition, which we denote by $- \parallel -$, one has the possibility to define programs composed of sequential threads that can execute concurrently on a shared memory. For instance, the program fragment $p \parallel q$ is now expressible:

$$x := 1 \ \parallel \ (x := 0 \ ; \ x := x+1)$$

where p and q are the two sequential programs defined above. We may wonder if the semantics of a parallel program is a function from initial values of the program variables (or initial stores) to final values of the program variables (or final stores) also in this enriched setting.

First observation: the result of the parallel program $p \parallel q$ is not unique, as the final value of x can be 1 or 2, depending on the actual execution ordering of the elementary assignments. In particular, 2 is the final value for x when the assignment of p is executed in between the two assignments of q. As we cannot make any assumption on the relative execution speed of p and q, we are to accept any possible intertwined ordering, so that both final values for x are to be considered admissible. Hence, the computation is *nondeterministic*. Nonetheless, it is possible to associate a function to $p \parallel q$, but a more complex function that associates to any possible initial store s (with arbitrary value associated to x) the two possible final stores $s[1/x]$ and $s[2/x]$; in general, such a function goes from initial stores to *sets* of final stores.

Second observation: program equivalence based on the identity of the computed functions is not a *congruence* for parallel composition, i.e., it is not preserved by the operator of parallel composition. As a matter of fact, we have noted that p and q are semantically equivalent; however, $p \parallel p$ and $p \parallel q$ are not equivalent, as they compute different functions: on the one hand, for $p \parallel p$ the final value of x can only be 1, no matter the ordering of the assignments they perform; on the other hand, for $p \parallel q$ the final value of x can be 1 or 2, depending on the actual ordering of execution.

Third observation: a consequence of the observation above is that no *compositional semantics* is definable over functions. As a matter of fact, a compositional semantics for parallel composition is definable only if a semantic operator $- \otimes -$ exists over functions, corresponding to the syntactic operator $- \parallel -$ on programs. Therefore, since the semantics of p and q is the same function f, the semantics of $p \parallel q$ should be $f \otimes f$, and the semantics of $p \parallel p$ should be $f \otimes f$ as well. However, we have already noted that $p \parallel p$ and $p \parallel q$ compute different functions and so $f \otimes f$ is not definable: the correct semantics for $p \parallel p$ and $p \parallel q$ cannot be computed when abstracting from the intermediate states of the computation, as the function-based semantics does by associating to p and q the same function f. Summing up, a compositional semantics for the parallel operator cannot be defined over functions.

Fourth observation: often a concurrent program is not meant to compute a result, but rather to offer a service, possibly forever; for instance, an operating system is a concurrent program that is assumed not to terminate. Therefore, a function is not an appropriate semantic tool for expressing the behavior of concurrent programs, as all the non-terminating programs would be equated, independently of the different services they offer.

In conclusion, the semantics of concurrent programs cannot be defined satisfactorily in terms of functions from initial values of the program variables to final values of the program variables.

1.1.2 Interaction, Communication and Process Algebra

If the semantics of a concurrent program is not a function, what is it? From the example above, we have understood that a concurrent program offers a much richer behavior:

- *Interaction* among different entities is possible and we should be concerned about *when and how* the program *interacts* with its *environment*, i.e., the outside world. For instance, program p and q above are not equivalent in the way they interact with the memory: contrary to p, program q interacts twice with the memory and the memory intermediate state (in between the two interactions) offers to the environment (i.e., to other programs interacting with the memory) the possibility to interact with x holding value 0; such an interaction capability of the memory is not possible when executing program p.
- *Nondeterminism* is often an inevitable effect of different relative speed of execution of independent threads. For instance, program $p \parallel q$ above is nondeterministic.
- *Non-termination* is often a desirable property of a concurrent system, as its duty is to offer a service that, in principle, should be available forever.

Therefore, in general, a concurrent system (or *reactive system*) is to be seen as a system that may react to stimuli from its environment and, in turn, influence its environment by providing feedback. A simple example of a reactive system is any coffee vending machine, where the environment is a customer interacting with it by inserting coins, selecting the kind of coffee, and, finally, receiving a cup of coffee from the machine. Other more complex examples of reactive systems include operating systems, communication protocols, software embedded in mobile phones, control systems for transportation (such as flight or railway control systems), and so on.

The key idea of interaction is crucially based on the assumption that all the involved entities should be considered *active*, i.e., they can compute autonomously and interact by *message-passing* (i.e., by synchronizing or communicating values) with the other entities. According to this intuition, data structures, as well as the memory variables, are not *passive* entities used by programs, rather they are to be considered as autonomous, active interacting entities, willing to communicate the value they store to any program requiring it, as well as willing to rewrite the value they store according to a write request by some interacting program.

Hence, the basic building blocks for the model of a reactive system should be the atomic, indivisible activities, i.e., the *actions*, the reactive system performs either in isolation or by interacting with the environment. The execution of an action determines a change in the current *state* of the system. Therefore, the suitable semantic model for a reactive system is a sort of state-transition automaton, called *labeled transition system*. (See Section 2.1 for an introductory example of how a coffee vending machines can be modeled by means of labeled transition systems.)

A labeled transition system modeling a real system can be enormously large, so that a manageable description of it is often mandatory. A *specification language* can be a good solution to this problem. Such a language:

- should be simple enough to be equipped with a clear, well-defined semantics in terms of labeled transition systems, so that it can provide linguistic support for describing succinctly such models, possibly in a compositional manner;
- should be expressive enough to be able to represent a large class of labeled transition systems, including at least all those with finitely many states and transitions;
- should be executable, so that the specification can be analyzed before being implemented (early prototyping);
- should provide support for a compositional analysis of the model.

Process algebras — such as CCS [Mil89], ACP [BK84a, BW90], CSP [Hoa85, Ros98], and Lotos [BoBr87, BLV95], just to mention a few — emerged about thirty years ago as good specification languages for reactive systems (see [Bae05] for a historical overview). They are composed of a minimal set of linguistic operators, equipped with simple and intuitive semantics, with the desire to single out a very basic formalism for concurrency. To partially explain the great variety of the different proposals, we should be aware that the basic operators of different process algebras are chosen according to different intuitions on the basic mechanism of computation. For instance, only considering the communication mechanism, we can recognize at least the following different features:

- *synchronous* vs *asynchronous*: the former when the send action and the receive one are performed by the interacting partners at the same time; the latter when the send action is decoupled from the receive one.
- *point-to-point* vs *multi-party*: the former when the involved partners are only two, the latter when several partners interact at the same time.

The possible four combinations may give rise to different process algebras with different expressive powers. For instance, we will see in Chapter 6 that synchronous point-to-point communication of CCS (also called *handshake* communication) and synchronous multi-party communication of CSP are not equally expressive.

The interesting features that a process algebra may possess can be dramatically diverse. They may include, besides many different forms of communication, also the aspects related to:

- forms of sequentialization (e.g., operators of *action prefixing* for CCS and of *sequential composition* for ACP);
- scoping of names (e.g., operators of *restriction* for CCS and of *hiding* for CSP),
- mobility (e.g., in the π-calculus [MPW92] channel names are communicable values, allowing for dynamic reconfiguration of the system),
- priority among actions (e.g., [CLN01, VBG09]),
- security (e.g., [RSG+, FG01, FGM02]),
- real time (e.g., [Yi91, NS94, HR95]),
- performance evaluation (e.g., [Hil96, BG98, H02]),

and so on. In some of the above cases, in order to equip the process algebra under investigation with a satisfactory semantics, the model of labeled transition systems needs to be enriched to include further information, e.g., about time. For an advanced overview on different aspects of process algebra we refer you to the *Handbook of Process Algebra* [BPS01].

The process algebra presented in this textbook is CCS, proposed by Robin Milner [Mil80, Mil89, Mil99]. It has been chosen mainly for its deep simplicity, elegance of its algebraic theory and good expressive power, and also because it has been extended smoothly to include other features, such as *mobility* with the π-calculus [MPW92, Mil99, SW01], *security* with SPA [FG01] and Crypto-SPA [FGM02], *real time* with TCCS [Yi91], and so on; however, such extensions are not discussed here.

1.2 Why This Book?

This section illustrates the main distinctive features of this book with respect to other books on process algebra, such as [Mil89, AILS07, BBR10, San12]. The intended reader of this section is an instructor (a person who already knows a lot about this theory), who may wish to know the pros (and cons) of this book.

The main motivation for this book is the adoption of a different methodological approach: this book first presents and discusses the semantic model, i.e., *labeled transition systems*, together with a variety of sensible behavioral equivalences over them; then it proposes suitable linguistic means to define objects of the semantic model, i.e., it proposes a process algebra able to express all the labeled transition systems of interest. As a matter of fact, a distinctive feature of this textbook is the presence of some *representability theorems*; for instance, one shows that all labeled transition systems with finitely many states and transitions can be represented, up to isomorphism, by processes of a subcalculus of CCS called finite-state CCS.

Another distinctive feature of this book is the discussion about the relative expressive power of subcalculi of CCS (and also of other process algebras in Chapter 5) and their precise relationship w.r.t. the well-known classification of formal languages (regular, context-free and context-dependent languages), as described in the Chomsky hierarchy (Section 1.3.3). Indeed, in many parts of the book emphasis is put on the similarities and differences w.r.t. the classical theory of automata and formal languages (see Sections 1.3.2, 1.3.3 and 1.3.4 for a short overview). Moreover, a student-level, detailed treatment of Turing-completeness within CCS is provided.

Expressiveness is also the key aspect in the study of encodability of additional operators in CCS. For instance, hiding and sequential composition are proved derivable in CCS. The proof of these results can be seen as a correctness proof of a compiler from a source language (typically CCS enriched with some additional operator) to a target language (typically CCS).

Expressiveness limitations of CCS are also investigated: even if CCS is Turing-complete, it cannot solve all the problems one may wish to solve in concurrency

theory. We will see that a deterministic, symmetric, fully distributed solution to the well-known dining philosophers problem [Dij71] cannot be provided in CCS, because of its limited synchronization discipline: binary, point-to-point (or handshake) communication. An extension to overcome this inability is presented in Chapter 6, where Multi-CCS is introduced by extending CCS with atomic behavior and multiparty synchronization.

1.2.1 Structure of the Book

Chapter 1 contains a brief introduction to concurrency theory, as well as a description of the structure of the book and some background material.

Chapter 2 introduces the semantic model we use throughout the textbook: labeled transition systems (LTSs for short). They are equipped with a suitable set of different behavioral equivalences, ranging from isomorphism to trace equivalence, the latter being very similar to the classic *language equivalence* over finite automata (see Section 1.3.4).

Chapter 3 presents the *Calculus of Communicating Systems*, CCS for short, in particular its syntax (which slightly differs from [Mil89]) and its operational semantics in terms of LTSs. A large part of this chapter is devoted to studying various subcalculi of CCS, in order to investigate their relative expressive power and algorithmic properties, and to offer a large collection of case studies of increasing complexity, ranging from basic examples of vending machines, to more complex examples of counters, stacks and queues. Of particular interest is *regular* CCS, as it corresponds to finite-state LTSs, as well as *finitary* CCS, whose programs are finitely representable. This latter calculus is the CCS subcalculus that is mainly used throughout the book; it is shown to be Turing-complete (see Section 1.3.4 for a definition of Turing-completeness), even if it cannot represent all the possible LTSs; moreover, all the behavioral equivalences studied in Chapter 2 turn out to be undecidable for finitary CCS, while they are all decidable for regular CCS. Finally, this chapter introduces a richer variant of CCS, called *value-passing* CCS, which explicitly allows for the communication of values; this variant is proved to be as expressive as CCS.

Chapter 4 discusses the algebraic properties of various behavioral equivalences: as CCS is built around a set of operators, such as parallel composition, it is natural to study which properties of such operators hold w.r.t. a given behavioral equivalence. For instance, parallel composition is associative and commutative w.r.t. all the equivalences. This chapter also investigates whether the equivalences of Chapter 2 are actually congruences w.r.t. the CCS operators. It turns out that most of the equivalences are indeed respected by the CCS operators. Finally we discuss the problem of *axiomatizing* such behavioral congruences, i.e., of finding a suitable set of axioms that characterize syntactically the behavioral congruence under investigation.

Chapter 5 shows that some useful operators, proposed in other process algebras, are actually derivable in CCS, so that CCS turns out to be a reasonably expressive language. A large part of this chapter is devoted to studying the ACP sequential composition operator, whose semantics needs a proper extension of the LTS model. An encoding of an extension of CCS, enriched with sequential composition, into CCS is proposed and proved correct.

However, even if Turing-complete, CCS is not able to model some additional useful behavior, such as the atomic execution of sequences of actions as well as multiparty synchronization. To this aim, Chapter 6 introduces an extension to CCS, called Multi-CCS, that is able to model these behavioral aspects. Some classical concurrency control problems, such as the *concurrent readers and writers* [CHP71], can be now solved satisfactorily in Multi-CCS, while they are not solvable in CCS.

1.2.2 How to Use It

Note for the instructor: The book is the result of several years of teaching a master's course in Concurrency Theory at the University of Bologna. The intended audience is composed of advanced undergraduate (or graduate) students.

The core of the book is composed of Chapters 2, 3 and 4, which can be a good basis for a semester course, possibly complemented with a lab with verification tools, such as the Concurrency Workbench [CWB]. Chapter 4 could be taught before the second part of Chapter 3 (about CCS subcalculi and case studies), in case the instructor wishes to fully develop the theory of CCS before discussing applications. Chapter 5 is useful if the instructor thinks it is a good idea to expose the students to the problem of encoding one language into another; indeed, the chapter offers some examples of this sort, from very basic to more advanced. Chapter 6 is useful if the instructor wishes to discuss limitations of the CCS language and a possible extension to overcome some of them. Technically, this chapter is more involved, as it presents more advanced techniques, such as working with a structural congruence; moreover, it introduces the reader to non-interleaving semantics, such as *step* semantics.

1.3 Background

This textbook is intended for an audience of students who have been already exposed to some introductory courses in mathematics and theoretical computer science, in particular, basic courses on discrete mathematics, formal languages and automata theory, as well as computability. In this case, this section can be skipped.

Nonetheless, this book can also be read by those who have very little knowledge of these topics, as we have tried to be as self-contained as possible. As a matter of

fact, this section introduces a few notions — some very basic, some more advanced — that are referred to in the text. Of course, their presentation is very succinct, with little explanation and few examples, and in no way is it intended to replace a thorough exposition that can be found in well-known textbooks completely devoted to these topics, such as [HMU01, Sip06, Koz97, FB94].

1.3.1 Sets, Relations and Functions

We assume the reader is familiar with the notion of set. Set $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{N} = \{0, 1, 2, \ldots\}$. Set $\mathbb{R}$ is the set of real numbers. Given a set A, we write $x \in A$ to mean that x is an *element* of A, and $x \notin A$ to mean that x is not an element of A. Given two sets A and B:

- their *union* is $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$;
- their *intersection* is $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$;
- $A - B = \{x \mid x \in A \text{ and } x \notin B\}$ is the set difference of A and B;
- $\overline{A} = \mathscr{U} - A$ is the complement of A w.r.t. the *universe* set $\mathscr{U}$, containing all the elements of interest;
- we write $A \subseteq B$ to mean that A is a *subset* of B, i.e., that each element of A is also an element of B; if this is not the case, we write $A \not\subseteq B$, i.e., there exists at least one element in A which is not an element of B. Moreover, $A \subset B$ if $A \subseteq B$, but $A \neq B$.

Note that for any set A, A is a subset of A and also the *empty set* $\emptyset$, the set with no elements, is a subset of A. The set of all the subsets of A, called the *powerset* of A, is denoted by $\mathscr{P}(A) = \{B \mid B \subseteq A\}$.

Given two sets A and B, the *Cartesian product* $A \times B$ is the set $\{(x,y) \mid x \in A \text{ and } y \in B\}$. An element (x,y) is called a *pair*. More generally, when the Cartesian product is among many sets, e.g., $A_1 \times A_2 \times \ldots \times A_k$, an element $(x_1, x_2, \ldots, x_k)$ is called a *tuple*.

Given $B = A_1 \times A_2 \times \ldots \times A_k$, a *relation* R is a subset of B, i.e., a set of tuples. A *binary relation* on a set A is a subset of $A \times A$. For instance, relation $S \subseteq \mathbb{N} \times \mathbb{N}$, defined as $S = \{(n,m) \mid \exists k \in \mathbb{N} \text{ such that } n \times k = m\}$, relates n and m if n is a divisor of m. Given a binary relation R, its *inverse* R^{-1} is the relation $\{(y,x) \mid (x,y) \in R\}$. For instance, $S^{-1} = \{(m,n) \mid \exists k \in \mathbb{N} \text{ such that } n \times k = m\}$, i.e., m is a multiple of n. The *identity relation* $\mathscr{I}$ on A is the relation $\{(x,x) \mid x \in A\}$.

Given two binary relations, R and S, on a set A, we define the *relational composition* $R \circ S$ as the set $\{(x,z) \mid \exists y \in A \text{ such that } (x,y) \in R \text{ and } (y,z) \in S\}$.[1]

A binary relation $R \subseteq A \times A$ is *reflexive* if $(x,x) \in R$ for all $x \in A$, i.e., if $\mathscr{I} \subseteq R$. Relation R is *symmetric* if whenever $(x,y) \in R$ then also $(y,x) \in R$, i.e., if $R^{-1} \subseteq R$.

[1] Our notation for the relational composition $R \circ S$ is not standard, as it is more customary to write the two arguments in reverse order: $S \circ R$. However, for the aims of our book, we prefer to adopt this order of arguments: $R \circ S$.

Relation R is *transitive* if whenever $(x,y) \in R$ and $(y,z) \in R$ then also $(x,z) \in R$, i.e., if $R \circ R \subseteq R$.

A relation R that is reflexive, symmetric and transitive is called an *equivalence relation*. Let $[a]_R$ denote the equivalence class of a w.r.t. the equivalence relation R, i.e., $[a]_R = \{b \in A \mid (a,b) \in R\}$; the set of its equivalence classes $\{[a]_R \mid a \in A\}$ determines a *partition* of A, i.e., $A = \bigcup_{a \in A} [a]_R$ and $[a]_R \cap [b]_R = \emptyset$ if $(a,b) \notin R$, while $[a]_R \cap [b]_R = [a]_R$ if $(a,b) \in R$.

A binary relation R on A is *antisymmetric* if for all $x,y \in A$, if $(x,y) \in R$ and $(y,x) \in R$ then $x = y$. R is a *partial order* if R is reflexive, antisymmetric and transitive. For instance, relation $S = \{(n,m) \mid \exists k \in \mathbb{N} \text{ such that } n \times k = m\}$ is a partial order: if $(n,m) \in S$ and $(m,n) \in S$, then there exist k_1 and k_2 such that $n \times k_1 = m$ and $m \times k_2 = n$; thus, $(m \times k_2) \times k_1 = m$, which is possible only if both k_1 and k_2 are 1, and so $n = m$, hence S is antisymmetric; moreover, S is trivially reflexive ($(n,n) \in S$ by choosing $k = 1$) and transitive (if $(n,m) \in S$, i.e., $n \times k_1 = m$, and $(m,p) \in S$, i.e., $m \times k_2 = p$, then $(n,p) \in S$ because $n \times k_1 \times k_2 = m$).

A relation R is a *preorder* if it is reflexive and transitive. For instance, relation T on a set B of persons, defined as $\{(x,y) \mid x \text{ is not taller than } y\}$, is a preorder; note that T is not a partial order: if x is as tall as y, then $(x,y) \in T$ and $(y,x) \in T$, but x and y are two different persons, i.e., $x \neq y$; also T is not an equivalence relation: if y is taller than x, then $(x,y) \in T$ but $(y,x) \notin T$, hence T is not symmetric.

Given a binary relation R on a set A, the *reflexive closure* of R is the relation R' such that: (i) $R \subseteq R'$, (ii) $\mathscr{I} \subseteq R'$, and (iii) R' is the least relation satisfying (i) and (ii) above. This can be formalized by saying that R' is the least relation satisfying the following inference rules:

$$\frac{(x,y) \in R}{(x,y) \in R'} \qquad \frac{x \in A}{(x,x) \in R'}$$

Such relation R' is simply $R \cup \mathscr{I}$. Note that any relation R'' such that $R' \subset R''$ satisfies the two rules, but it is not the least one.

Given a binary relation R on a set A, the *symmetric closure* of R is the relation R' such that: (i) $R \subseteq R'$, (ii) $R'^{-1} \subseteq R'$, and (iii) R' is the least relation satisfying (i) and (ii) above. Such a relation R' is simply $R \cup R^{-1}$.

Given a binary relation R on a set A, the *transitive closure* of R is the relation, denoted by R^+, such that: (i) $R \subseteq R^+$, (ii) $R^+ \circ R^+ \subseteq R^+$, and (iii) R^+ is the least relation satisfying (i) and (ii) above. This can be formalized alternatively by saying that R^+ is the least relation satisfying the following inference rules:

$$\frac{(x,y) \in R}{(x,y) \in R^+} \qquad \frac{(x,y) \in R^+ \;\; (y,z) \in R^+}{(x,z) \in R^+}$$

For instance, if $A = \{x,y,z,t\}$ and $R = \{(x,y),(y,z),(z,t),(y,x)\}$, then $R^+ = R \cup \{(x,z),(x,x),(y,t),(y,y),(x,t)\}$. Note that relation $R' = R^+ \cup \{(z,z)\}$ satisfies the two rules, but it is not the least one.

Given a binary relation R on a set A, the *reflexive and transitive closure* of R, denoted by R^*, is $R^+ \cup \mathscr{I}$. This can be equivalently formalized as the least relation satisfying the following rules:

$$\frac{(x,y) \in R}{(x,y) \in R^*} \qquad \frac{x \in A}{(x,x) \in R^*} \qquad \frac{(x,y) \in R^* \;\; (y,z) \in R^*}{(x,z) \in R^*}$$

A binary relation $R \subseteq A \times B$ is a *function* if for all $x \in A$ there exists *exactly one* $y \in B$ such that $(x,y) \in R$; in such a case, we use notation $R : A \to B$ where set A is called the *domain* and set B the *codomain* of R. We usually use letters $f, g, \ldots$ (or mnemonic names) to denote functions and we write $f(a) = b$ to express that $(a,b) \in f$; in such a case b is called the *image* of a under f. As an example, $double : \mathbb{N} \to \mathbb{N}$, defined as $double(n) = 2 \times n$, is a function associating to each number n in the domain, the even number $2 \times n$ of the codomain. Relation $R \subseteq A \times B$ is a *partial function* if for all $x \in A$ there exists *at most one* $y \in B$ such that $(x,y) \in R$. In such a case, we use notation $R : A \multimap B$.

As functions are relations, we can define their composition as we did for relations. Given two functions $f : A \to B$ and $g : B \to C$, their *composition* $f \circ g : A \to C$ is the function that associates to each $a \in A$ the value $g(f(a)) \in C$. The definition of functional composition scales also to partial functions in the obvious way.

A function $f : A \to B$ is *injective* (or *one-to-one*) if for all $x, y \in A$, $f(x) = f(y)$ implies $x = y$; equivalently, f is injective if $\forall x, y \in A$, $x \neq y$ implies $f(x) \neq f(y)$. For instance, function *double* above is injective. Function f is *surjective* (or *onto*) if for all $b \in B$ there exists an $a \in A$ such that $f(a) = b$, i.e., each element of B is in relation with at least one element of A. For instance, function *double* above is not surjective, because odd numbers are not the image of any number. Function f is *bijective* if it is both injective and surjective. For a bijective function $f : A \to B$, we can define its *inverse* $f^{-1} : B \to A$ as follows: $f^{-1}(b) = a$ if and only if $f(a) = b$.

A set A is *finite* if there exists $n \in \mathbb{N}$ and a bijective function $f : A \to \{1, 2, \ldots, n\}$. In such a case $|A| = n$ denotes the *cardinality* of A, i.e., the number of elements in A. A set A is *denumerable* if there exists a bijective function $f : \mathbb{N} \to A$. For instance, the set of even numbers $P = \{n \mid n = 2 \times k, k \in \mathbb{N}\}$ is denumerable: the required bijective function is function $double(n) = 2 \times n$. A set A is *countable* if it is either finite or denumerable. A set A is *uncountable* if there is no bijective function $f : \mathbb{N} \to A$. For instance, set $\mathbb{R}$ is uncountable; this can be proved by using Cantor's *diagonalization method* (see, e.g., [Sip06] for a detailed account of this method).

A *multiset* (or *bag*) M over a set A is an unordered, possibly infinite, list of elements of A, where no element of A can occur infinitely many times. It can be represented formally as a function $M : A \to \mathbb{N}$ such that $M(x)$ is the number of instances of element $x \in A$ in M. Given two multisets M_1 and M_2 over the set A, we write $M_1 \subseteq M_2$ if $M_1(x) \leq M_2(x)$ for all $x \in A$. A multiset M over A is *finite* if $M(x) > 0$ for only finitely many $x \in A$. Of course, if A is finite, then any multiset over A is a finite multiset. The set of all finite multisets over a set A is denoted by $\mathscr{M}_{fin}(A)$.

1.3.2 Alphabets, Strings, Languages and Regular Expressions

An *alphabet* A is a finite, nonempty set, e.g., $A = \{a, b\}$. An element a of A is called a *symbol*. A *string* (or *word*, or *trace*) over A is any finite length sequence of symbols of A, e.g., *aaa* or *abbba*. The *empty string* ε is the string composed of no symbols. We use w, x, y, z, possibly indexed, to represent arbitrary strings.

The *length* of w, denoted as $|w|$, is the number of occurrences of symbols in w; e.g., $|aaa| = 3$, $|\varepsilon| = 0$ and $|abbba| = 5$. The power of a symbol, say a^n, denotes a string composed of n occurrences of a. This can be defined inductively as follows:

$$a^0 = \varepsilon \quad \text{and} \quad a^{n+1} = aa^n.$$

Set A^* is the set of all the strings over alphabet A. For instance, if $A = \{a\}$, then $A^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\} = \{a^n \mid n \in \mathbb{N}\}$. Given any alphabet A, set A^* is countable, as it is possible to define a bijective function $f : \mathbb{N} \to A^*$. Intuitively, we can list all of its strings by first enumerating all the strings of length 0 (only ε), then those of length 1 (all the symbols of A, which we can assume to be ordered in some alphabetical order), followed by those of length 2 (which are ordered in the lexicographical order induced by the alphabetical order), and so on. We denote by A^+ the set $A^* \setminus \{\varepsilon\}$, i.e., the set of all the nonempty strings over A.

The *concatenation* of strings x and y is the string xy obtained by juxtaposition of the two. This operation is associative and the empty string is its neutral element:

$$(xy)z = x(yz) \qquad x\varepsilon = x = \varepsilon x.$$

Moreover, $|xy| = |x| + |y|$ and $a^n a^m = a^{n+m}$. The power x^n of a string x is the juxtaposition of n copies of x: $x^0 = \varepsilon$ and $x^{n+1} = xx^n$.

If $a \in A$ and $x \in A^*$, we write $\sharp(a, x)$ for the number of occurrences of a in x; for instance, $\sharp(a, abbba) = 2$, $\sharp(b, abbba) = 3$ and $\sharp(c, abbba) = 0$.

A *prefix* of x is any initial substring of x; formally, y is a prefix of x if there exists z such that $yz = x$. A *sufffix* of x is any final substring of x; formally, y is a suffix of x if there exists z such that $zy = x$.

A *language* L is any subset of A^*, $L \subseteq A^*$. For instance, if $A = \{a\}$, any of the following subsets of $A^* = \{a^n \mid n \in \mathbb{N}\}$ is a language: $\{a, aaaa\}$, A, A^*, $\{a^n \mid n = 2 \times k, k \in \mathbb{N}\}$, $\emptyset$, $\{\varepsilon\}$. Note that $\emptyset$ is the empty language, i.e., no string belongs to $\emptyset$, while $\{\varepsilon\}$ is a one-string language.

A language L is *prefix closed* if whenever $xy \in L$, then $x \in L$, for all $x, y \in A^*$. Hence, if L is prefix-closed, then $\varepsilon \in L$ or $L = \emptyset$.

Languages can be concatenated: $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$. For instance, if $L_1 = \{a, aaa\}$ and $L_2 = \{b, bb\}$, then $L_1 \cdot L_2 = \{ab, abb, aaab, aaabb\}$; moreover, if $L_3 = \{a^n \mid n = 2 \times k, k \in \mathbb{N}\}$, then $L_1 \cdot L_3 = \{a^n \mid n = 2 \times k + 1, k \in \mathbb{N}\}$.

We can define the power L^n of a language L as follows: $L^0 = \{\varepsilon\}$ and $L^{n+1} = L \cdot L^n$. The *iterate* (or Kleene star) L^* and the *positive iterate* L^+ of a language L are defined as follows:

$$L^* = \bigcup_{n \geq 0} L^n \qquad L^+ = \bigcup_{n \geq 1} L^n$$

Note that since A is a language, the set A^* of all strings over A is such that $A^* = \bigcup_{n \geq 0} A^n$, as required by the definition above. Note also that $\emptyset^* = \{\varepsilon\}$ because $\emptyset^0 = \{\varepsilon\}$.

Regular expressions over an alphabet A, ranged over by e (possibly indexed), are defined by means of the following syntax in Backus-Naur Form (BNF):

$$e ::= \mathbf{0} \mid \mathbf{1} \mid a \mid e + e \mid e \cdot e \mid e^* \mid (e)$$

where a is any symbol in A. The syntax above is ambiguous: for instance, $a + b \cdot c$ can be interpreted as $(a+b) \cdot c$ or $a + (b \cdot c)$. To solve this problem, we assume that the operators have a different binding strength: the iterate postfix operator * binds tighter than the binary infix concatenation operator $\cdot$, in turn binding tighter than the binary infix alternative operator $+$. With this convention, $a + b \cdot c$ represents $a + (b \cdot c)$. Moreover, the concatenation operator is often omitted, so that $e_1 \cdot e_2$ is simply denoted as $e_1 e_2$. Examples of regular expressions are: ab^*, $a + \mathbf{0}$, $(a+b)^*$ and $a(a+b)c^*$. Regular expressions are used to denote languages as follows:

$$\begin{array}{lll} \mathscr{L}[\mathbf{0}] = \emptyset & \mathscr{L}[e_1 + e_2] = \mathscr{L}[e_1] \cup \mathscr{L}[e_2] & \mathscr{L}[(e)] = \mathscr{L}[e] \\ \mathscr{L}[\mathbf{1}] = \{\varepsilon\} & \mathscr{L}[e_1 \cdot e_2] = \mathscr{L}[e_1] \cdot \mathscr{L}[e_2] & \mathscr{L}[e^*] = (\mathscr{L}[e])^* \\ \mathscr{L}[a] = \{a\} & & \end{array}$$

For instance, regular expression $a \cdot b^*$ denotes the language $\mathscr{L}[a \cdot b^*] = \mathscr{L}[a] \cdot \mathscr{L}[b^*] = \{a\} \cdot (\mathscr{L}[b])^* = \{a\} \cdot (\{b\})^* = \{a\} \cdot \{b^n \mid n \in \mathbb{N}\} = \{ab^n \mid n \in \mathbb{N}\}$.

A language L is *regular* if there exists a regular expressions e such that $L = \mathscr{L}[e]$. For instance, $L = \{a^n \mid n = 2 \times k + 1, k \in \mathbb{N}\}$ is regular because $L = \mathscr{L}[a(aa)^*]$.

1.3.3 Grammars and the Chomsky Hierarchy

Languages can be generated by means of *grammars*. A *general* grammar G is a tuple (N, T, S, P), where N is a finite set of *nonterminals* (ranged over by capital letters $A, B, C, \ldots$), T is a finite set of *terminals* (the symbols of the alphabet), $S \in N$ is the initial nonterminal and P is a finite set of *productions* of the form $\gamma \to \delta$, with $\gamma, \delta \in (T \cup N)^*$.

The language generated by a general grammar G, denoted by $L(G)$, is given by the set of all strings (or words) $w \in T^*$ derivable by rewriting from S: formally, $L(G) = \{w \in T^* \mid S \longrightarrow^* w\}$, where $S \longrightarrow^* \gamma$ is the minimal relation induced by the following axiom and inference rule:

$$\frac{}{S \longrightarrow^* S} \qquad \frac{S \longrightarrow^* \alpha\gamma\beta \qquad \gamma \to \delta \in P}{S \longrightarrow^* \alpha\delta\beta} \text{ where } \alpha, \beta \in (T \cup N)^*$$

The form of the productions can be restricted in some way, thus yielding a classification of grammars, called the *Chomsky hierarchy*.

A grammar G is *right-linear* if all of its productions are of the form $B \to a$ or $B \to bC$ or $B \to \varepsilon$, where $B,C \in N$ and $a,b \in T$. For instance, grammar $G_1 = (\{S,A\},\{a,b\},S,\{S \to aA, A \to bA, A \to \varepsilon\})$ is right-linear and its generated language $L(G_1) = \{ab^n \mid n \in \mathbb{N}\} = \mathscr{L}[ab^*]$. It can be proved that the class of languages generated by right-linear grammars coincides with the class of regular languages, i.e., the class of languages denoted by regular expressions.

The intersection of two regular languages is a regular language. For instance, $\mathscr{L}[ab^*] \cap \mathscr{L}[a^*b] = \{ab^n \mid n \in \mathbb{N}\} \cap \{a^nb \mid n \in \mathbb{N}\} = \{ab\} = \mathscr{L}[ab]$, which is a finite language, hence regular.

A grammar G is *context-free* if all of its productions are of the form $B \to \gamma$, with $B \in N$ and $\gamma \in (T \cup N)^*$. A typical example of a context-free grammar is $G_2 = (\{S\},\{a,b\},S,\{S \to aSb, S \to \varepsilon\})$, which generates the language $L(G_2) = \{a^nb^n \mid n \in \mathbb{N}\}$. A language L is context-free if there exists a context-free grammar G such that $L(G) = L$. The class of context-free languages includes the class of regular languages because a right-linear grammar is also a context-free grammar. Such inclusion is strict because there are context-free languages that are not regular, e.g., language $L(G_2)$.

The intersection of a context-free language with a regular language is a context-free language. For instance, $L(G_2) \cap \mathscr{L}[(aa)^*(bb)^*] = \{a^{2n}b^{2n} \mid n \in \mathbb{N}\}$, which is context-free.

A context-free grammar G is in *Greibach normal form* if all of its productions are of the form $B \to a\beta$, where $a \in T$ and $\beta \in N^*$. In case ε belongs to the language to be generated by grammar G, it is admitted an ε-production $S \to \varepsilon$ for the initial nonterminal S, provided that S never occurs on the right-hand-side of any production. It can be proved that for any context-free grammar G, there exists a context-free grammar G' in Greibach normal form such that $L(G) = L(G')$.

A grammar G is *context-dependent* (or *monotone*) if all of its productions are of the form $\gamma \to \delta$, with $\gamma,\delta \in (T \cup N)^+$ and $|\gamma| \leq |\delta|$. As above, an ε-production $S \to \varepsilon$ for the initial nonterminal S is allowed, provided that S never occurs on the right-hand side of any production. A typical example of a context-dependent grammar is $G_3 = (\{S,B\},\{a,b,c\},S,\{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\})$, which generates the language $L(G_3) = \{a^nb^nc^n \mid n \geq 1\}$. A language L is context-dependent if there exists a context-dependent grammar G such that $L(G) = L$. The class of context-dependent languages includes the class of context-free languages because a context-free grammar in Greibach normal form is also a context-dependent grammar. Such inclusion is strict because there are context-dependent languages that are not context-free, e.g., language $L(G_3)$.

1.3.4 Finite Automata and Turing Machines

A *nondeterministic finite automaton* (NFA, for short) M is a tuple (Q,A,δ,F,q_0) such that Q is a finite set of *states*, A is a finite alphabet of *input symbols*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* of type $\delta : Q \times (A \cup \{\varepsilon\}) \to \wp(Q)$.[2]

A configuration is a pair (q,w) with $q \in Q$ and $w \in A^*$. Configurations can evolve according to the following rules:

$$\frac{}{(q,w) \longrightarrow^* (q,w)} \qquad \frac{(q,w) \longrightarrow^* (q',\sigma w') \qquad q'' \in \delta(q',\sigma), \sigma \in A \cup \{\varepsilon\}}{(q,w) \longrightarrow^* (q'',w')}$$

An NFA $M = (Q,A,\delta,F,q_0)$ recognizes (or accepts) a string $w \in A^*$ if there exists a final state $q \in F$ such that $(q_0,w) \longrightarrow^* (q,\varepsilon)$, i.e., there is a path starting from the initial state q_0 in the automaton that, by reading w, leads to a final state. The automaton M recognizes the language $L[M] = \{w \in A^* \mid \exists q \in F.(q_0,w) \longrightarrow^* (q,\varepsilon)\}$.

A *deterministic finite automaton* (DFA, for short) M is an NFA (Q,A,δ,F,q_0) such that $\delta(q,\varepsilon) = \emptyset$ and $|\delta(q,a)| = 1$ for all $q \in Q$ and $a \in A$. In other words, δ has type $\delta : Q \times A \to Q$. Therefore, DFAs are a subclass of NFAs. However, from an expressiveness point of view, they are equivalent: given an NFA M, we can construct a DFA M' such that $L[M] = L[M']$ (by means of the *Rabin-Scott subset construction* [RS59]).

The class of languages recognized by finite automata coincides with the class of regular languages.

Informally, a (deterministic) *Turing machine* is composed of a finite control, which can be in any of a finite number of states, and a tape, of unbounded length, which is divided into cells; each cell can hold one symbol from a given alphabet. A tape head, which is positioned at one single tape cell, can read the content of that cell or write onto it. Initially, the input is written on the tape, while all the other cells hold a special symbol called *blank*, and the tape head is positioned at the leftmost symbol of the input. At each stage of the computation, the Turing machine reads the symbol in the cell pointed to by the tape head and, depending also on the current state, it writes a symbol in that cell, transits to the next state and moves the tape head — left or right — to the next cell. If the Turing machine eventually stops in a final state, then the input is accepted. If it stops in a non-final state, then the input is rejected. But it may also never end its computation for a given input. Hence, a Turing machine M computes a partial binary function $g_M : A^* \multimap \{0,1\}$, where $g_M(w) = 1$ if w is accepted by M, $g_M(w) = 0$ if M stops on w in a non-final state, but $g_M(w)$ may be undefined when M never ends, i.e., it is unable to accept/reject w.

Turing machines can be adapted to compute partial binary functions on any countable set B, notably $\mathbb{N}$. Turing machines can also be adapted to compute par-

[2] Function δ can be equivalently defined as a subset of $Q \times (A \cup \{\varepsilon\}) \times Q$, i.e., as a set of triples of the form (q,a,q') for $q,q' \in Q$ and $a \in A \cup \{\varepsilon\}$.

tial functions from $\mathbb{N}$ to $\mathbb{N}$. A function $f : \mathbb{N} \multimap \mathbb{N}$ is *Turing-computable* if there exists a Turing machine that computes it. A formalism is *Turing-complete* if it can compute all the Turing computable functions. A few examples of Turing complete formalisms are the *lambda calculus* [B84], *Counter Machines* (see Section 3.5.1), as well as any programming language that includes **while**, **if-then-else**, assignments and sequential composition [BJ66]. Hence, function f is Turing-computable if there exists an algorithm that computes it.

1.3.5 Decidable and Semi-decidable Sets and Problems

Given a set $B \subseteq \mathbb{N}$, its *characteristic function* $f_B : \mathbb{N} \to \{0,1\}$ and its *semi-characteristic function* $g_B : \mathbb{N} \multimap \{1\}$ are defined as follows:

$$f_B(x) = \begin{cases} 1 & \text{if } x \in B, \\ 0 & \text{if } x \notin B \end{cases} \qquad g_B(x) = \begin{cases} 1 & \text{if } x \in B, \\ \text{undefined} & \text{if } x \notin B \end{cases}$$

Set B is *decidable* (or *recursive*) if its characteristic function f_B is Turing-computable, i.e., there exists an algorithm that can compute such a function. A few examples of decidable sets are: $\emptyset$ ($f_\emptyset$ is the constant function $f_\emptyset(x) = 0$ for all $x \in \mathbb{N}$), $\mathbb{N}$ ($f_\mathbb{N}$ is the constant function 1), any finite subset of $\mathbb{N}$, $P = \{n \mid n = 2 \times k, k \in \mathbb{N}\}$. Set B is *undecidable* if its characteristic function f_B is not Turing-computable.

Set B is *semi-decidable* (or *recursively enumerable*) if its semi-characteristic function g_B is Turing-computable, i.e., there exists an algorithm that can compute such a function. Of course, any decidable set B is also semi-decidable, because the algorithm computing the characteristic function f_B can be easily adapted to compute the semi-characteristic function g_B. Examples of semi-decidable sets that are not decidable are less easy to find. Suppose $Z_0, Z_1, Z_2 \ldots$ is an enumeration of Turing machines. Then, $B = \{x \mid Z_x \text{ with input } x \text{ terminates}\}$ is semi-decidable as its semi-characteristic function g_B is Turing-computable. It can be proved that B is undecidable, as its characteristic function f_B is not Turing-computable: this is related to the so-called *halting problem*, discussed below. An example of a non-semi-decidable set is $C = \{x \mid Z_x \text{ with input } x \text{ does not terminate}\}$.

B is *effectively decidable* if it is decidable and the algorithm, computing the characteristic function f_B, can be explicitly exhibited. An example of a decidable but not effectively decidable set can be constructed as follows. Take $A = \{0\}$ and the non-semi-decidable set C described above; then consider $A \cap C$, which can be either A itself, in case $0 \in C$, or the empty set, in case $0 \notin C$. Of course, both A and $\emptyset$ are effectively decidable with their associated algorithms, so for sure there exists an algorithm that computes $f_{A \cap C}$ (i.e., $A \cap C$ is decidable), but we are unable to decide which of the two algorithms is the right one, because we are unable to test $0 \in C$, hence $A \cap C$ is not effectively decidable.

A language $L \subseteq A^*$ is a countable set, hence we can define characteristic and semi-characteristic functions for it. It can be proved that context-dependent languages are all decidable (or recursive), but there exist decidable languages that are not context-dependent. The class of languages generated by general grammars coincides with the class of semi-decidable (or recursively enumerable) languages.

A problem can be usually seen as a particular function, and sometimes such a problem is decidable (or *solvable*) if and only if its corresponding function is computable. For instance, the famous *halting problem* for Turing machines can be formulated as the following function *halt*. Given an enumeration of Turing machines, $Z_0, Z_1, Z_2, \ldots$, function $halt(x,y)$ — where x is an index of a Turing Machine and y is an input — is defined as

$$halt(x,y) = \begin{cases} 1 & \text{if } Z_x(y) \text{ terminates} \\ 0 & \text{otherwise.} \end{cases}$$

Solving the halting problem for Turing machines means being able to compute function *halt*. Unfortunately, function *halt* is not computable and so the halting problem is unsolvable. If *halt* were computable, then we could compute also function $K(x) = halt(x,x)$. If function K were computable, we could even compute function G, defined as

$$G(x) = \begin{cases} 1 & \text{if } K(x) = 0 \\ \text{undefined} & \text{if } K(x) = 1. \end{cases}$$

But now since G is computable, there should exist a Turing machine that computes G; suppose Z_j is this Turing machine. Then, we get a contradiction when computing $G(j)$: either $G(j) = 1$, which is possible only if $K(j) = 0$ and so $Z_j(j)$ must diverge, contradicting the fact that $G(j)$ returns 1; or $G(j)$ is undefined, which is possible only if $K(j) = 1$ and so $Z_j(j)$ converges (also a contradiction). As the only assumption we make in our reasoning is that function *halt* is computable, we can conclude that *halt* is not computable, and so the halting problem is undecidable.

Another well-known problem is the *membership problem*: given an enumeration $G_0, G_1, G_2, \ldots$, of a class of grammars over an alphabet A and given an enumeration $w_0, w_1, w_2, \ldots$, of strings in A^*, we want to decide if $w_i \in L(G_j)$ for all $i, j \in \mathbb{N}$. The problem has an obvious associated function *Mem*, defined as

$$Mem(x,y) = \begin{cases} 1 & \text{if } w_x \in L(G_y) \\ 0 & \text{otherwise.} \end{cases}$$

The computability of function *Mem* depends on the considered class of grammars: it is computable for context-dependent grammars, while it is not computable for general grammars. Therefore, the membership problem is solvable for context-dependent grammars while it is not for general grammars.

There are also problems whose associated functions are always computable. For instance, given an enumeration of programs $p_0, p_1, p_2, \ldots$, for a given programming language, and given a decidable predicate P over such programs, the *existential P problem* is solvable if there exists an index i such that $P(p_i)$ is true. Formally, given the computable function f_P, defined as

$$f_P(x) = \begin{cases} 1 & \text{if } P(p_x) \text{ is true} \\ 0 & \text{otherwise,} \end{cases}$$

the function $exist_P$ associated to the existential P problem is defined as

$$exist_P(x) = \begin{cases} 1 & \text{if } \exists i \text{ such that } f_P(i) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Function $exist_P$ is computable for any programming language because it is either the constant function 1 or the constant function 0.[3] However, we say that the existential P problem is solvable for a given formalism only if $exist_P$ is the constant function 1.

[3] To be precise, such a function is computable, but may be not *effectively* computable, i.e., an algorithm does exist for sure, but we may be unable to exhibit it explicitly.

http://www.springer.com/978-3-319-21490-0

Introduction to Concurrency Theory
Transition Systems and CCS
Gorrieri, R.; Versari, C.
2015, XI, 334 p. 63 illus., Hardcover
ISBN: 978-3-319-21490-0