Chapter 2
Language, Logic and Computations

> *Realizing that the aliens would not understand us, we were continuously sending out the Pythagorean Theorem and other simple geometrical propositions. But our call into space remained without an answer.*
>
> Stanisław Lem, *Magellan's Cloud*

It seems difficult to define mathematics. A possible definition, or rather an explanation of mathematics, could be that mathematics is an extension of our language that enables us to perform rigorous deductions. The problem with this explanation is that it does not take into account the theorems and proofs that are the main products of this field. Nevertheless, the role of mathematics as a means of expressing our ideas precisely is unquestionable. It is also possible to observe the growing number of mathematical terms in common language. Of course, many of these terms were present before the advent of mathematics and they just obtained a more precise meaning in the course of the development of mathematics. The first thing that comes to ones mind are numerals. The etymology of numerals 11 and 12 in many languages witnesses that the decimal system had been accepted at their early stage. (The most likely etymology, say, of the English word *eleven* is '*one left over*'.) There are many geometrical concepts commonly used, for example, *triangle, square, line, curve, cylinder*. There are also many more modern concepts, like *function, minimum, set, probability*. This is not the privilege of mathematics; scientific terms from all fields penetrate common language. Mathematics is special in that it is applied in other sciences, but not conversely, other sciences can provide problems and motivation for mathematics, but cannot be used there.[1] Because of its universal nature, mathematics has been proposed as a communication means with extraterrestrial civilizations.

A language is not merely a collection of words. The words must have some *meaning*. A very primitive language, perhaps a language of some species of animals, may consist of words and their meaning, without any complicated constructions. Human languages allow combinations of words to talk about arbitrarily (arbitrarily at least in principle) complicated things. This requires some rules, rules that say how we

[1] There is one exception: computer science is used in experimental mathematics.

P. Pudlák, *Logical Foundations of Mathematics and Computational Complexity*,
Springer Monographs in Mathematics, DOI 10.1007/978-3-319-00119-7_2,

can combine words into sentences—*the syntax*, and rules by which we deduce the meaning of sentences—*the semantics*. Furthermore, there is a special, and very basic, type of meaning of a sentence, its *truth*. This brings us to logic, the art of putting pieces of truth together in order to get new evidence.

'*Mathematics as the foundations of logical reasoning*' sounds good, but we are looking for the foundations *of mathematics* in this book. So let's try to invert it: *logic as the foundations of mathematics*. There is no doubt that logic is essential for mathematics. But do we know precisely what logic is? Or is the logic that we are using the right one? We definitely must learn more about logic to be able to answer such questions. Anticipating what we should learn in this chapter, let me say that it turns out that the logic that we use in mathematics is a very clear and unambiguous concept. There is no problem with logic as far as the foundations are concerned. It does not mean that it is a simple thing, on the contrary, there are computationally unsolvable problems in logic and a lot of problems for mathematical research. A different question is if logic suffices for the foundations. At the turn of the 20th century, several philosophers and mathematicians tried to find the foundations of mathematics based only on logic. This stream in the philosophy of mathematics is called *logicism*. The incompleteness theorems of Gödel proved that this goal (or at least its original version) cannot be achieved. We have to introduce specific axioms about sets in order to be able to develop mathematics. We will see how much we have to add to logic in Chap. 3 that deals with set theory.

Having a formal language and formal rules for logic one can do certain operations mechanically. '*Mechanically*' is an obsolete term; the 20th century's version should be '*electronically*', but we know what it means: one can find an algorithm, or put differently, write a program for that task. Such a task is, in particular, *proof checking*. To check the correctness of a formal proof does not require any intellectual ingenuity, it can be done by a machine. This fact has practical applications, but we are not interested in them. For us, it documents that the concept of a proof is so clear that one can check mechanically whether or not a given text is a correct proof. There are, however, tasks that are not algorithmically solvable, in particular the problem of finding a proof of a given sentence cannot be mechanized. Thus we have arrived at the concept of a computation, another concept that is very relevant for foundations. And again, the same questions: what is it, can it be precisely defined, etc? We'll see.

2.1 The Language of Mathematics

Now we will leave aside the use of mathematics as a language, and talk about the language used in mathematics. Let us look at a typical mathematical text. Nowadays it is written most often in English, but this is the least important property, as it can be translated into any language. The first most striking feature of a mathematical text is its very restricted vocabulary. Potentially it may contain any words and often words that are used in everyday life for things and properties that have nothing to do with mathematics, (for example, *ring, field, group, good, dense*) which, however, have a completely different meaning in mathematics. The reason for that is that sometimes people use analogies from real life to explain mathematical ideas. If mathematicians wrote papers for computers, whose life is restricted to solving given problems, the vocabulary would be even smaller. The grammar of mathematical texts is also only rudimentary. For instance, mathematics uses verbs only in a very restricted way. Surely, the verb *to be*, is used, but it only connects a noun with an adjective, or it is used to express existence. There are several other verbs that you can find in mathematical texts, for example, *suppose, assume, follow, imply*. Those can be avoided using a different sentence construction; we can replace them by connectives (namely *if*... *then*). In real life time is very important and thus a lot of the grammatical structures concern time. Most verbs express changes in time. Verbs come with various tenses expressing the relation to the present. Mathematics, on the other hand, expresses facts independent of time. Even if the problem concerns time we consider it as an entity; for example, instead of motion mathematicians consider its trajectory. Time, in the form we perceive it, is not present even in physics. It is just the fourth dimension. Mathematicians are like historians: they do not have to think what will happen, they do not have to perform experiments, they have the complete history of the phenomenon at hand. Like historians only need the past tense, mathematicians only need the present.[2]

So what remains of the natural language in mathematics? Here is what is used:

- nouns (*number, point, set,* ...);
- adjectives (*straight, continuous, large,* ...);
- prepositions and verbs expressing relations (a point *on* a line, a function *vanishes* at zero, ...);
- connectives (*and, or, not,* ...);
- two quantifiers '*there exists*' and '*for all*'.

Nouns are necessary because we should be able to speak about things. Things in mathematics are elements of structures, structures and sets. Typically we study a particular mathematical structure and then the things that we consider are elements of the structure. But also when speaking about structures and sets, we can always assume that they are elements of a larger structure, say, the structure of all sets. Hence nouns are elements.

Adjectives express properties. A property is something that concerns one element. Then we have relations, which, in the simplest case, concern pairs of elements and are called *binary relations*. Interestingly, there is no single word class that corresponds to relations. In most cases relations are expressed by prepositions or verbs, but they can also be expressed by certain combinations of words. From the point of view of logic, these are linguistic nuances and we put all relations into one class. Moreover, logicians also include properties in the same class and call them '*unary relations*'. Another word that is sometimes used for properties is '*predicates*'. This

[2] There are languages in which grammatical structure for expressing time is only rudimentary (for example, Chinese), other languages may lack other structures. The language of mathematics is certainly the poorest of all.

is, however, confusing because grammatically 'predicate' refers to a particular part of a sentence which, from the point of view of logic, may describe a binary relation like the following sentence:

Point P lies on line ℓ.

According to grammar '*lies on line ℓ*' is a predicate, according to logic the sentence expresses that a certain binary relation holds between a point and a line. To see the reason for thinking about the sentence as expressing a relation, notice that one can say the same as follows:

Line ℓ goes through point P.

The last two items on the list above concern logic. There are various ways of expressing the same connectives and the same quantifiers. Furthermore, the interpretation of connectives in a natural language is often a little different from the one of formal logical. The connective *or* is usually interpreted as the *exclusive* or, hence in law one often uses '*and/or*' to express the non-exclusive or. In logic *or* is the non-exclusive, which means that '*A or B*' is true also when both *A* and *B* are true; but the *exclusive or*, a different connective, is sometimes used too. In some languages and in slang, double negation is used just to stress simple negation, while in logic two negations cancel each other. All the connectives used in a natural language reduce to the following four *not, and, or, if... then*. The two quantifiers can also be expressed in many ways. For the universal quantifier, there are essentially the same expressions in all languages, which are words such as *all, every, any, each*. These words differ slightly in their use and meaning, but in logic, again, they are considered to be synonyms. For the existential quantifier, some languages have special constructions such as '*there is*', '*il y a*', '*es gibt*', but the most common way is simply to use the verb *to be*.

Natural languages also use *modalities*, which are words expressing how much we believe in what we are saying. Such words include *surely, necessarily, maybe, probably*. Modalities can be considered as unary connectives, similarly as we treat negation as a unary connective. Their meaning is not precise, according to the mathematical standards, therefore they are not used in mathematics.[3] But it does not mean that mathematics is completely deprived of the possibility of expressing such things. Consider the modalities *probably, likely, unlikely*, for example. Mathematics has developed a whole theory in order to be able to express such statements, the probability theory. Mathematicians, of course, do not use these vague words, instead they quantify numerically the probability of an event. Sometimes this is accepted also in our everyday life. For instance, in the USA they often forecast that the probability of rain is so many percent, instead of just saying that it is likely, unlikely, as they do elsewhere.

Similar is the role of the various quantifiers that are present in natural languages such as *a few, some, many, almost all*. In fact, it is one of the main goals of mathematics to replace these imprecise expressions by precise statements. This is achieved

[3] Modal logics are not used to state and prove mathematical results, but they have interesting applications, see provability logic on page 297.

by *counting*, which means using numbers to say precisely, or to estimate the quantity in question. Thus what remains from the language are the precise quantifiers *for all* and *there exists*.

It is important to realize that the possibility of replacing modalities and imprecise quantifiers by precise statements is the main reason why the former ones are not used in mathematics. Once in a while suggestions are made to enrich mathematics by modalities, other quantifiers, etc. These are futile proposals; this will never catch up because it is against the spirit of mathematics!

The aforementioned list is the result of a superficial analysis based on grammatical categories, thus one key item is missing on the list: *variables*. In the traditional classification there is no word class corresponding to variables, although it is one of the most important concepts in mathematics. Still, variables are present in the everyday use of language. Suppose for instance that you want to say that there are two elements without referring to the numeral 2. You can say: '*There is an element and there is another one different from the previous one*.' With three elements we would use something like: '*... and yet another one...*'. With four elements it will become rather messy. Here we have considered only the task of expressing the existence of a certain number of elements, but we have to handle the same problem in other situations where we need to talk about several elements. In a natural language we have very limited means for that; we have words like '*another, yet another, this, that*' which are good for two elements, and may be used for three, but for more they are not practical. What one uses then are descriptions like '*the tall man, the blond girl, the one who came first*'. However, the most efficient way is to use names. The most ancient mathematical texts used the awkward way of distinguishing elements of natural language. The simple trick of assigning letters as temporary names to the investigated entities, used already in antiquity, must have had dramatic consequences for the development of mathematics. It enabled mathematicians to treat much more complex problems than before.

Nowadays we call the temporary names *variables* and the permanent ones *constants*. Variables are most often x, y, z; examples of constants are π and e.[4] The number of variables that may be needed in an average length paper is surprisingly large. Therefore, people use various alphabets, indices, primes, and other marks.[5]

There is one more class, *functions*, which I will explain later.

You see that linguists study languages from a different perspective; grammatical categories do not render the substance of the language of mathematics. The correct classification for the language of mathematics is the following:

- constants and variables;
- functions;
- relations;

[4] The term *variable* is often used with another meaning, namely as a function; for example, a *random variable* is a function defined on a probability space.

[5] In one of my papers I used all lower case Latin letters and on top of that several Greek letters and some upper case Latin letters. This is not unusual.

- connectives;
- quantifiers.

This is the classification of mathematical logic, the field of science that studies the language of mathematics.

Why Is the Language of Mathematics so Restricted?

I spoke about *what* the language of mathematics looks like. Now the difficult question is '*Why?*'. Why do we use this particular fragment of our natural language? Why don't we use more? Does it have to be a fragment of a natural language? These are very difficult questions, yet we can give at least some partial answers.

Consider, for example, the modalities '*maybe, probably, surely, necessarily*'. These words sound alien to mathematicians, since in mathematics statements are either true or false, there is no third possibility, *tertium non datur*. Thus the question can be rephrased as: '*why only two truth values?*'. The answer is simple: *because two suffice and one is not enough.*

It is obvious that we have to use at least two truth values. The reason for using only two is not so much economy; the true reason is that we want to analyze the concepts as much as possible in order to understand their essence. Whenever it is possible to decompose a phenomenon into simpler components we do so and it always helps us. Thus physicists decompose molecules into atoms, in order to find laws about molecules, then they decompose atoms into elementary particles, in order to understand atoms better, etc. Concerning truth values, we cannot say that three truth values can be 'decomposed' into two, but we can describe all arguments of non-classical logics using just the classical one, the logic with two truth values.

Before going on I have to make a terminological digression. When new trends in the foundation of mathematics first appeared, such as intuitionistic mathematics, a term was needed to denote the standard approach. Thus we use phrases '*classical mathematics*' and '*classical logic*'. This is not very fortunate, since meanwhile a lot of modern mathematical fields appeared and one would hesitate to call such mathematics classical. In logic it is, however, very common to use this word to distinguish the standard approach from various alternatives, especially when talking about alternative logical calculi. Therefore I will stick to this word, (but remember, 'classical mathematics' includes almost all modern fields).

There are various ways we can translate uncertainty, fuzziness etc. into classical logic. For instance, for uncertain statements we can consider all possible cases where the uncertainty is replaced by either truth or falsehood. In the case of *fuzzy sets*, sets where elements are contained in the set with various values between 0 and 1, we can think of them simply as functions in the classical set theory.

The two values are not a dogma that is imposed on each new generation of mathematicians, it is simply the most convenient way of thinking. Another reason for classical two valued logic is its uniqueness among minimal possible logics. The connectives are not simply *some* connectives that can be used in two valued logic. By taking combinations, the connectives of classical logic generate all possible Boolean functions.[6] It would be much harder to agree on a non-classical logic, as there are many and there is no natural choice among them. The usual argument for someone who proposes a non-classical logic is that their logic contains the classical one and on top of that has other means of expressing statements and making deductions. But when they are describing the "new" logic, they use classical logic. So using classical logic the new system can always be described as another mathematical structure.

I used the truth values only as an example of a general principle. What mathematicians actually do is more general: they look for the simplest universal system that suffices for a given purpose. In the problem considered here, it is the language of mathematics. I argued that two truth values are the right choice for truth values. There are more thing to decide, for example, quantifiers. We need them in order to be able to talk about unspecified entities, to be able to make generalizations and eventually to be able to talk about infinity. One quantifier suffices, say, the quantifier '*for all*'. The existential quantifier '*there exists*' is dual to '*for all*', and it is definable from it. Hence, once we take one of them we actually accept also the other one. But we do not need more! If we want to quantify in another way, we can use mathematical structures such as probability spaces, numbers, etc.

Such a reductionistic trend in science can be documented by many examples. The axiomatic method is a good example: a theory, for example, a physical theory, is considered better if based on smaller number and simpler equations. We think it is better, not because we save space and time for its presentation, but because it *explains better* the studied phenomenon, because it is *more likely to be true*, or simply because it *looks nicer*. We prefer to explain a certain phenomenon by referring to a single principle, rather then to several.

Is Logic Simply a Part of Natural Language?

In order to explain the language of logic, I presented it as a fragment of a natural language, but I do not claim that natural language is primary and logic is derived from it. A language is necessary for logic, but, in a sense, logic is absolute. There are many natural languages and there are huge differences between languages that are not related, such as the difference between English (or other Indo-European languages) and Chinese (dialects). But there is no difference between the logics used by the people using different languages—their logic is the same.

When I say that language is necessary for logic I mean it in a very broad sense. The language does not have to be a human language. There are several species of very intelligent animals that only use a few sounds for communication between themselves. So their 'vocabulary' is very small and does not contain essentially any logic. But they are able to perform deductions. The language that they are using

[6] We use conjunction, disjunction, implication and negation, but it suffices to only take one of the three binary plus negation.

for this purpose is an internal language of each individual, the language of *concepts* that they are able to form in their brains. I am sure animals think very much like we do, namely, they imagine possible situations, in particular situations that may occur after they perform a particular action, and choose the one that is the most favorable for them. Intelligence is the ability to see similarities between situations and thus be able to better estimate the consequences. Logic is a means of organizing this process.

The Language of Mathematical Logic

Mathematical logic is the field that deals with the language used in mathematics, the proofs and the truth of mathematical statements. Therefore, we will call a formal language for mathematics a *logical language*. Later I will also talk on deduction, so we will also have some rules to derive true sentences. Such systems are called *logical calculi*.

People often associate mathematical logic with symbols (and often it is the reason why they do not like it). The role of symbols in logic is emphasized more than needed. At early stages the name *symbolic logic* was used to distinguish mathematical logic from its sister branch in philosophy. But symbols are not any more important in logic than in any other branch of mathematics. Symbols enable us to express things more compactly, more precisely and sometimes they are useful for calculations. But in spite of this and in spite of the tendency of mathematicians towards brevity and precision, logic symbols are almost never used in mathematical writings with the exceptions of mathematical logic itself and set theory.

The same can be said about the syntactical rules, the rules describing how formulas are constructed. One of the first things that you encounter in a textbook on logic is a lengthy and boring description of the syntax. But unless you are going to study a particular formal system, you do not need it. The syntax of a formal logical language is only a simplified syntax of a natural language. Nevertheless, there is a good reason for presenting it so formally in textbooks. What is needed is to show that a formal language can be presented precisely and thus studied as a mathematical concept. A natural language is a rather vague and complicated thing, moreover, it is changing in time; it may be anything except a precise mathematical concept. Mathematicians, who use language as a tool not as a subject of their study, do not mind that the language is not precisely defined. But if you want to state and prove a theorem concerning language, you need something precise. Therefore, you need to do the boring job of explicitly writing down all the syntactical rules.

However, it depends on the person studying logic; if somebody is willing to accept that a formal concept of a language can be constructed, they do not need to consider an example of such a formalization. Once the primitives are described, it is not necessary to describe how a sentence is formed from these primitives as it is essentially the same as in natural languages. The actual syntax may vary for different logical languages, but this is not important. There are a few things that one has to secure; the main thing is to structure sentences in such a way that they allow only unique reading, which is usually done using parentheses. In natural languages we use pauses in speech instead of parentheses, and punctuation in writing; also there are some words for separation. But it does not work so perfectly, there are sentences that can be interpreted ambiguously, (which is a problem well-known among lawyers).

Actually, a large part of the contemporary population has a very intimate experience with formal languages. Those are the people who know a programming language. For such computer minded readers, I do not have to describe a logical language at all. Logical languages have been prototypes for programming languages, the difference being only what the texts in the languages expresses: statements in logic, algorithms in a programming language.

In order to complete the picture of the logical analysis of the language of mathematics, we need some more detail and, after all, I also have to mention at least the basic symbols used in mathematical logic.

The description in the last section was simplified, as we did not consider an important type of concepts, which are functions. Functions are present in natural languages too, for example, '*the mother of*' is the function that assigns to a person his or her mother. Though functions can be considered to be only a special kind of relations, it is more practical to use them as primitives. Functions are the main subject of studies in many branches of mathematics. The corresponding syntactical concept in logic, the names of functions, are called *function symbols*.

Having function symbols we get a new type of syntactical concept the *terms*. A term is an expression formed from function symbols, variables, and constants. What children use already at elementary school as *algebraic expressions* are terms that use function symbols for the basic arithmetical operations. Terms are, of course, used in other structures too. The interpretation of a term is a function if the term contains variables, or an element of a structure if it only contains constants. For example, $1+1$ is one of the names for the number 2, while $2x+1$ denotes a linear function. For binary functions, one often uses *infix* notation and calls them operations; this concerns mainly $+$ and $\times$ and other group operations. Again, for a programmer the concept of a term is very familiar, as this part of logic completely penetrated programming languages. No wonder, functions are here to be computed.

Now, having terms and a relation symbol, we can form an *atomic* formula, which is a formula that contains no logical symbols, namely, no connectives and no quantifiers. For example, take terms $x+2$ and $x \cdot y$ and the binary relation symbol $<$ and form an atomic formula $x+2 < x \cdot y$.

In algebra we often need only one relation, the relation of equality. Equality is often treated as a logical primitive, but again this is only a matter of convenience. It is a relation that occurs very frequently, so it is more practical to assume that it is present in logic. Furthermore, it is interesting to study just equations (expressions of the form $s = t$, with s, t terms). Already this fragment of logic is sufficiently complex.

The next step is to combine atomic formulas using connectives and quantifiers. Taking the above atomic formula and, say, $x = z^2$, we can form, for example, the conjunction of the two $x+2 < x \cdot y \wedge x = z^2$.

The role of quantifiers is very important, so let's consider some examples of their use.

All people are good.

This is a sentence from life, so the variable is not denoted by a letter. The variable there is hidden in *people*, which at the same time specifies the range of the variable. This sentence has the same structure as:

Every number is prime.

This sentence is false, but this is a positive fact: we are able to decide its truth value. Now take:

A number is prime.

Here we cannot decide, if it is true or false. The variable *number* is not quantified, it is *free*. There are two more possibilities to make a sentence with a definite truth value. One is to use the existential quantifier:

There is a prime number.

The other is to substitute a concrete element for the variable, say:

5 *is prime.*

Bertrand Russell explained the meaning of formulas with free variables as being *propositional functions*. As when we write $\sin x$, we denote a function that can have any value between -1 and 1, so the sentence *x is prime* can have either of the two values *true* or *false*. In fact, one of the names for the logical calculus with predicates, variables and quantifiers used to be the *functional calculus*. This denotation is not used anymore; we call it *first-order logic.*

Theoretically we could avoid formulas with free variables, but it is quite convenient to use them inside of proofs. For example, a proof in number theory may start with:

Let x be a number. Suppose x is prime...

But a theorem must be true under all circumstances, hence all variables must be *bound* by quantifiers. Therefore, in logic we reserve the word '*sentence*' for formulas in which all variables are bound and hence have a definite truth value.

The quantifiers are usually denoted by ∀ (for All) and ∃ (there Exists). Quantifiers bound variables exactly in the same way as, say, summation and integration operators. For example, consider a binary function $\sin(x + y)$. When we integrate along x, say, $\int_0^1 \sin(x + y)\, dx$, we get a function of only one variable y. We say that x is bound in $\int_0^1 \sin(x + y)\, dx$. A similar thing happens if we take a formula $\varphi(x, y)$ with two free variables and apply, for instance, the universal quantifier to x. The resulting formula is $\forall x\ \varphi(x, y)$. While the truth of $\varphi(x, y)$ depended on x and y, the truth of $\forall x\ \varphi(x, y)$ only depends on y.

Single quantifiers or several quantifiers of the same kind are easy to understand. The complexity arises when we alternate the two kinds. One alternation, such as

$\forall x \exists y\ \varphi(x, y)$, can be understood fairly easily. In this way we can express things like the infinitude of the prime numbers:

For every *prime*, there exists *a larger prime*.

To grasp the meaning of $\forall x \exists y \forall z\ \psi(x, y, z)$ is not so easy. In natural speech it is never used. When it is needed it is somehow circumvented. Consider, for example, the sentence:

In every *town* there is *the tallest building*.

There are only two quantifiers in this sentence, but the third one is hidden in '*the tallest*'. The latter fact is expressed using the comparative relation *taller* as follows:

The building taller than every *other building*.

Plugging this definition into the last sentence we get a sentence with three alternating quantifiers, which shows a typical way of avoiding several alternations of quantifiers in speech. Three alternating quantifiers occur in the definition of the limit of a function. This is one of the first definitions that students encounter when starting with the calculus. The inability to grasp the meaning of such a definition is used to sort out those who do not have a talent for mathematics. But the concept of the limit can be explained, not quite precisely, without the three quantifiers as follows. y_0 is the limit of the function f at point x_0, if

whenever x is very close *to x_0, then $f(x)$ is* very close *to y_0*.

The quantifiers are hidden in the imprecise expression '*very close*'. (It is possible to develop a theory in which this apparently vague expression has a precise meaning, see Chap. 3, *Robinson's Nonstandard Analysis and Vopěnka's Alternative Set Theory*.)

To imagine the meaning of four alternations of quantifiers seems as difficult as to imagine four-dimensional space. But there is a way out: using the concept of a game we can imagine even long alternations of quantifiers. I will explain it in the next chapter.

Notes

1. *The language of propositional logic.* This language uses

   a. an infinite set of propositional variables,
   b. connectives, and
   c. parenthesis.

   The standard connectives are in Table 2.1.

   All connectives can be defined using one of the first three and negation. Other connectives are also used sometimes; e.g., XOR (exclusive or) and propositional constants: true ($\top$, 1) and false ($\bot$, 0).

   In intuitionistic logic only equivalence can be defined from the other connectives.

Table 2.1 Standard connectives

Symbols	English word	Name
$\wedge$, &	and	conjunction
$\vee$	or	disjunction
$\rightarrow$, $\supset$	if ... then	implication
$\neg$, $\sim$	not	negation
$\equiv$, $\Leftrightarrow$	if and only if	equivalence

2. *The language of first-order logic.* This language has two parts: logical and nonlogical. The *logical symbols* are the symbols of propositional logic except for propositional variables. Further, it uses (first-order) variables and quantifiers $\forall$ (universal) and $\exists$ (existential). In classical logic it suffices to use one quantifier.

   The *nonlogical symbols* are constant symbols (which may be treated as 0-arity function symbols), function symbols and relation symbols of various arities. A typical theory uses a finite set of non-logical symbols. I will present an example after I define context-free languages. Equality ($=$) is usually treated as a logical symbol, but it can also be used as a special binary relation.

3. *Context-free languages.* The theory of formal languages was founded by Noam Chomsky in the 1950s. I will start with an important concept from this theory, *context-free languages*. It is worth noting that the concept was discovered when studying natural languages. Nowadays it is an important concept in computer science. In logic it is not used, but the way logicians define a formal language of a logical calculus is equivalent to a use of a context-free language. I will use a definition of logical formulas based on a context-free language as it shows a connection between natural languages and formal ones.

   In the mathematical language theory a *language* is a very general concept, it is any set L of finite strings of symbols from a finite set A. The set A is called the *alphabet* of the language L; the strings are called *words*.

   To define a context-free language we need another finite set N, disjoint with A, whose elements are called *nonterminal symbols*, with one distinguished nonterminal symbol s. Furthermore we need a finite set of pairs consisting of a nonterminal symbol and a word in the alphabet $A \cup N$. Such a pair (a, w), $a \in N$, w a word, is usually written as $a \Rightarrow w$, indicating that a can be replaced by w, and it is called a *rewrite rule*. The set of such rules is called a context-free *grammar* of the language. (Using general rewrite rules one defines general grammars, but we do not need them here.) The language determined by the grammar is the set of all words that can be obtained starting with the initial symbol s and applying rewritings.

   The terminology that I am using here is common in computer science, but does not apply to natural languages. When considering natural languages we should call A the *vocabulary* and the strings of symbols *sentences*. The nonterminal symbols are *grammatical categories* such as *subject, predicate, object, adverbial clause*. The initial symbol s represents the class of all sentence in the language. Let us consider a couple of rules that are probably valid for any human language.

$$\begin{aligned}&<\text{sentence}> \Rightarrow <\text{subject}><\text{predicate}>\\&<\text{predicate}> \Rightarrow <\text{verb}><\text{adverbial clause}>\end{aligned}$$

The meaning of the angled brackets $<\cdots>$ is that the compound expression denotes a single symbol. Thus <sentence>, <subject>, etc. are elements of the set of nonterminal symbols. If we only were interested in the structure of sentences, we would use only the grammatical categories such as *noun, verb, adverb*, etc. as terminal symbols. If we want to get actual sentences, we need also rules that transform nonterminal symbols into terminal ones, such as

$$\begin{aligned}&<\text{verb}> \Rightarrow\ abandon\\&<\text{verb}> \Rightarrow\ abase\\&<\text{verb}> \Rightarrow\ abash\\&\quad\dots\end{aligned}$$

The context-freeness means that we are describing grammatically correct sentences, with no regard to their meaningfulness. So sentences such as '*A yellow poem lies in the air.*' are considered to be in the language. In fact, probably no natural language is context-free because there are always some dependencies between the forms of words due to declension, conjugation, etc., that cannot be completely described by context-free rules.

Programming languages are usually defined as context-free languages, but there are often additional exceptions that spoil this property. The famous programming language ALGOL, created soon after the birth of the formal language theory, was based on a context-free grammar.

4. *The language of first-order logic, cont'd.* We will consider a language with two connectives $\wedge, \neg$ ('and' and 'not', which suffice to define all other connectives), universal quantifier $\forall$ ('for all', the other quantifier, $\exists x\dots$' is definable by $\neg\forall x\neg\dots$), equality $=$ and an infinite set of variables (say $x, x', x'', x''', \dots$). In our simple example, there is one binary relation symbol R, one binary function symbol F and a constant c.

   A context-free grammar for this language has three nonterminal symbols: <formula>, <term> and <variable>, with <formula> being the initial symbol. The rules are

$$\begin{aligned}&<\text{formula}> \Rightarrow\ \forall <\text{variable}> (<\text{formula}>)\\&<\text{formula}> \Rightarrow\ (<\text{formula}>)\wedge(<\text{formula}>)\\&<\text{formula}> \Rightarrow\ \neg(<\text{formula}>)\\&<\text{formula}> \Rightarrow\ R(<\text{term}>,<\text{term}>)\\&<\text{term}> \Rightarrow\ F(<\text{term}>,<\text{term}>)\\&<\text{term}> \Rightarrow\ <\text{variable}>\\&<\text{term}> \Rightarrow\ c\\&<\text{variable}> \Rightarrow\ <\text{variable}>'\\&<\text{variable}> \Rightarrow\ x\end{aligned}$$

This system has superfluous parentheses around atomic formulas, which can be avoided by having a nonterminal symbol for atomic formulas and a few more rules.

5. *Higher-order languages.* In a second-order language we have variables for relations and for functions. Usually we distinguish first-order symbols from second order symbols by using lower case letters for first-order symbols and upper case letters for the second-order symbols. This is not enough for all the bookkeeping that one would need if all done quite formally. To this end we have to declare for each second-order symbol whether it is a relation or a function and then of what number of variables. Note that the relations and functions of the first-order language can be viewed as first-order constants in higher order languages.

   As examples, I will write formally the axiom of induction and an axiom of topological spaces.

   a. Suppose we are describing the natural numbers, thus all elements are natural numbers. Then we do not have to mention the set $\mathbb{N}$ and the axiom of induction reads:

   $$\forall X\big(\big(X(0) \wedge \forall x\big(X(x) \rightarrow X(x+1)\big)\big) \rightarrow \forall x\ X(x)\big).$$

   b. To write down the second axiom of the two axioms of topological spaces, I will use the same symbol $\mathcal{O}$ for the predicate expressing that a set is open. Thus the predicate $\mathcal{O}$ is a third order constant. I am using capital calligraphic letters for third order objects. Then the axiom reads:

   $$\forall \mathcal{X} \forall Y\big(\big(\forall X\big(\mathcal{X}(X) \rightarrow \mathcal{O}(X)\big) \wedge \forall x\big(Y(x) \equiv \exists X\big(\mathcal{X}(X) \wedge X(x)\big)\big)\big) \rightarrow \mathcal{O}(Y)\big).$$

6. *Propositional logic.* Propositional logic is the part of logic that uses neither quantifiers nor equality. Then the structure of atomic formulas does not matter; the only thing that matters is which atomic formulas are the same. Thus we can use any symbols for atomic formulas, preferably we use *propositional variables*. There is a good reason for referring to them in this way, as their meaning is simply *true* or *false*. We can view propositional logic as the study of a two element structure. The two elements represent *true* and *false* and they are usually denoted by 1 and 0. On this structure we study operations (but not relations). The operations are called *Boolean functions*. Boolean functions corresponding to negation, conjunction, disjunction, etc. are defined by the familiar truth tables.

   The idea of studying propositional logic as the theory of a two element set is due to George Boole (1815–1864) and therefore we talk about Boolean functions, Boolean algebras, etc. Gottfried Wilhelm Leibniz (1646–1716) was very close to this discovery. He noticed that, when interpreting true as 1 and false as 0, conjunction is the product. He thought that disjunction should be the sum, but that did not work. That was before mathematicians realized that it is not necessary to stick to familiar structures and that one can invent new interesting ones.

   A set of operations is called *a complete set of connectives*, if every operation on $\{0, 1\}$ can be expressed using operations from the set. For example, $\{\neg, \wedge\}$ and $\{\neg, \vee\}$ are complete. All these are simple facts, but they are important for realizing that at least propositional logic is uniquely determined: *it is the theory of the simplest nontrivial set.*

7. *Normal forms.* Many problems on formulas become simple if we can use a normal form, that is, if we can transform a general complicated formula into a formula having a nice structure. Everybody knows that (due to the distributive law) it is possible to write any polynomial as a sum of products of variables and constants (called *monomials*). A similar fact holds for propositional logic, where we have two distributive laws ($x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$), and De Morgan's laws $\neg(x \wedge y) \equiv (\neg x \vee \neg y)$ and $\neg(x \vee y) \equiv (\neg x \wedge \neg y)$. This enables us to write every proposition as a disjunction of conjunctions of propositional variables or negated propositional variables. This is called the *Disjunctive Normal Form* or simply DNF. There is, of course, the dual version, the *Conjunctive Normal Form*, or CNF. Note that the interpretation of a DNF is that we list all cases when the formula is true. This is not a very efficient way of expressing a given Boolean function, in fact, the reduction to a DNF or CNF often results in an exponential blow-up in the size. Thus DNFs and CNFs simplify problems in propositional logic only theoretically.

   In first-order logic we also have a nice normal form. First we move all quantifiers to the beginning of the sentence. This is enabled by rules such as $\neg\forall x\ \phi \equiv \exists x \neg\phi$ and $\phi \wedge \forall x\ \psi \equiv \forall x\ (\phi \wedge \psi)$, where x does not occur in ϕ. Then the inner part of the formula contains no quantifiers, thus we can transform it into a DNF (or a CNF). The resulting formulas are called *prenex normal forms*. This is a useful and efficient reduction, but the prenex normal forms are often more difficult to understand than the original sentences where the quantifier occur in places to which they actually refer.

   Having all quantifiers in a prefix enables us to define the *quantifier complexity* of sentences. Rather than counting the number of quantifiers we count the *number of alternations* of quantifiers. Furthermore, it is important to know what is the first quantifier in the prefix. If the number of alternations is small we denote the class simply by listing the quantifiers.

   Example $\exists x \forall y \forall z \exists u\ \phi$, with ϕ quantifier-free, is a $\exists\forall\exists$ formula.

   If we have more alternations, we write only the first quantifier indexed by the number of alternations. So the above formula is a $\exists_3$ formula. Sometimes people use Σ and Π instead of $\exists$ and $\forall$, but that may lead to confusion with other hierarchies.

8. *Equational theories.* I spoke about natural languages as a motivation for the language of first-order logic. This concerns propositional connectives and quantifiers. Function symbols, terms and equations come from mathematics. Function symbols describe elementary operations, terms describe computations and equality is a basic binary relation. In first-order logic we call equations atomic formulas. It may seem that they are too simple to be of any interest, but the contrary is true. Using equations one can express quite a lot, in fact we can, in some sense, simulate the whole first-order logic. To be quite precise we should note that a connective and a quantifier is implicit in equational theories. When we talk about a set of equations we mean, in fact, the conjunction of the equations.

When we say that an equation with variables is true, we mean that it is true for any possible value of the variables, which means that we implicitly assume that the variables are universally quantified. (For example, we state the commutative law as $x + y = y + x$, meaning that $\forall x \forall y(x + y = y + x)$.)

An important example of an equational theory is Boolean algebras. This is the equational theory of the two element set {0, 1}. We can take all operations that can be defined on this set (that is, all Boolean functions), or only a finite complete set of them. Thus propositional logic can be viewed as the equational theory of a two element set.

9. *Communication with extraterrestrials.* Mathematics would certainly be useful, but it is naive to expect that use of mathematical language would automatically solve the problem of communication. This problem was studied by the Dutch mathematician Hans Freudenthal. In his book *Lincos: Design of a Language for Cosmic Intercourse*, he presented a language for communication with extraterrestrials. His idea is roughly as follows. In order to be able to communicate with intelligent beings, we need a common language, but we cannot agree on a common language because it is impossible to exchange messages. Therefore we have to *design* a suitable language such that we can *teach* the other party this language. He proposed to teach by examples, starting with concepts from number theory, logic and set theory. When they learn the language, they will understand any messages that we send them.

   When designing messages for aliens the first thing one should do is to realize what we want to achieve. If the message should only convey that we are intelligent creatures, we do not have to send the Pythagorean Theorem, as the mere fact that we are able to send electromagnetic signals proves that our knowledge exceeds such trivial theorems. In such a case we only need to send signals that can be distinguished from those naturally occurring in space. A more difficult task is to persuade a potential recipient about our achievements in science (other than understanding electromagnetic waves), in particular, about our successes in mathematics. An especially interesting problem, but rather theoretical one, is how to persuade someone about having advanced computing technology. Problems of this kind have been studied in computational complexity theory in the case of the two parties exchanging information in both directions. In the situation when the recipient is hundreds of light years away, one has to assume only unidirectional communication and thus the problem is different.

   Naturally, it is more promising to look for incoming signals, but a number of signals have also been sent out.

2.2 Truth and Models

The Definition of Truth and Satisfaction

To define the concept of truth in general is a difficult philosophical problem. In mathematical logic, however, there is a precise definition of this concept. Truth is a relation between sentences and reality. I have described “mathematical reality” in the first chapter; it is the realm of mathematical structures. In the previous section I explained the formal language of mathematics as studied in mathematical logic. So the definition of truth is the definition of a certain relation between these two things. More precisely, it is a definition of the relation that ‘*a sentence ϕ is true in a structure M*’. It is more common to say that ‘*a sentence ϕ is satisfied in a structure M*’ and reserve the word ‘*truth*’ for a special situation that I will describe shortly. Thus we rather talk about the definition of *satisfaction*.

The definition of satisfaction is based on defining the meaning of the parts of the sentence. When we decompose a sentence, which is a formula in which there are no free variables, we get parts that do have free variables.

Example Consider the sentences

‘*For all* x*,* $x \leq 1$*, or* $x^2 > x$’

which is true in the natural numbers. This sentence contains a subformula

‘$x \leq 1$*, or* $x^2 > x$’

in which the variable x is free. Therefore we have to define satisfaction also for formulas and particular values of their free variables. In our example, we first define the satisfaction of the subformula for $x = 0, 1, 2, 3, \ldots$ and then the satisfaction of the sentence.

Given a formula $\phi(x_1, \ldots, x_n)$ a structure M and elements $a_1, \ldots, a_n$, the definition of satisfaction of ϕ by the elements $a_1, \ldots, a_n$ in M proceeds inductively, starting with atomic subformulas. We define the satisfaction of atomic formulas according to the relations and functions in M. The satisfaction of compound formulas is defined by interpreting connectives and quantifiers in the natural way. The formal definition is in Notes; here I will only consider an example.

Example The formula above has two atomic subformulas $x \leq 1$, and $x^2 > x$. The first one is satisfied by 0 and 1, otherwise it is not satisfied. The second one is satisfied for $2, 3, \ldots$. The subformula ‘$x \leq 1$*, or* $x^2 > x$’ is satisfied for every natural number because: for 0 and 1 the first term is true, for $2, 3, \ldots$ the second term is true, and a disjunction of two formulas is satisfied if at least one of them is. The sentence ‘*For all* x*,* $x \leq 1$*, or* $x^2 > x$*.*’ is true in the natural numbers because the subformula ‘$x \leq 1$*, or* $x^2 > x$’ is true for every natural number.

At first glance, this looks like a circular definition because we are defining satisfaction assuming that we already know what it is. We are defining ‘*for all* x’ by saying that it holds for all x. Certainly, if somebody did not understand the sentence, the definition would not help them to understand it. The Polish logician Alfred Tarski (1901–1983), who invented this definition, made this point by saying:

The sentence ‘It’s snowing.’ *is true if it’s snowing.*

So what is the matter? In order to understand this definition, one has to realize two things. Firstly, we are not in the position of philosophers who want to find the *meaning* of the concept of truth. We are defining truth and satisfaction as a *mathematical* concept. Forget about meaning and look at it as a mathematical definition.[7] There are sentences on one side and structures with their elements on the other. Since the sentences are formalized as certain strings, we are defining a relation between finite strings representing formulas and strings of elements of a structure. Thus this is a perfectly legitimate mathematical definition that can be formalized in set theory. Also it is a general definition that works for every structure and tells us in which structure a sentence is true and in which it is false.

It is instructive to consider the special case of finite structures. In this case, one can even write a program to determine, if a given sentence is true in a given structure. Programming languages often contain at least part of the propositional logic, so the task is simpler if we use such a language. The quantifiers will be tested by searching all elements of the structure, using constructs such as `do ... while ...`. Note that when writing such a program we are doing essentially the same as what we did above. In particular, we are programming how to test that a formula is satisfied for all x by letting the computer check it for all x. Because we are considering finite structures, there is no doubt that it makes sense—the computer will be able to tell us whether or not the formula is satisfied. Now, imagine an ideal computer that is able to do infinite computations. Then the definition of satisfaction for general structures can be viewed as a program for such a computer.

Secondly, we have to realize is that there are two levels of discourse. The lower level is the formal language for which we are defining the concept of satisfaction; it is called *the object language*. The upper level is the language that we use to make this definition; it is called *the metalanguage*. We have already observed that not distinguishing between the two levels leads to paradoxes, which would result in contradictions in formal systems. On the other hand, having this distinction, the definition makes sense: although we are using the same logical operators, such as '*or*' and '*there exists*', they appear in different places. In particular, we are defining '*or*' in the object language using '*or*' in the metalanguage. We suppose that we understand '*or*' in the metalanguage, so we can use it to define it in the object language.

The psychological factor that makes this definition difficult to accept is that we are defining something that is completely clear to us. Thus it seems that there is nothing to define. Therefore, we should view it as a *formalization* rather than a definition.

Let us now fix some terminology. Instead of saying that a sentence ϕ is satisfied in a structure M, one often says that *M is a model of ϕ*. This is further extended to theories. We say that M is a model of a theory T if all axioms of T are satisfied in M. We also often say '*models*' instead of '*structures*'. *Model theory,* an important

[7] And read the quotation from Isaac Asimov's *Imaginary* at the beginning of the next chapter (page 157).

field of logic, studies mathematical structures, that is, models, from the point of view of logic. The concepts of truth and satisfaction are the basic notions in this field.

When stating a theorem we often assume that the particular structure is clear from the context. For example, if we state that an arithmetical sentence is true, we mean that it is satisfied in the natural numbers. Surely, there are many other structures in which we can interpret the sentence.

This leads us back to philosophical questions. Although we do have a formal definition of truth, the meaning of this concept is a matter of philosophical views. Saying that a sentence is true presupposes an objective reality where the sentence should be satisfied. But what is mathematical reality? Specifically, are mathematical structures real? If not, how can we then talk about mathematical truth? Another question is, assuming we believe in mathematical reality, how do we acquire mathematical knowledge and how do we learn what is true? We cannot empirically test sentences that talk about an infinite number of elements. We can only decide the validity of a sentence in small finite structures. For large infinite structures, as well as for large finite ones, we use proofs instead of empirical tests. But proofs need axioms; logic alone does not suffice. So we need to justify axioms. How do we justify axioms if we cannot test their validity? And so on…

I leave these questions without an answer for the time being. What I am going to present further in the book should give us more ideas on which we can base our opinion. Finally, in Chap. 7, I will address these questions directly.

Logically Valid Sentences

When talking about truth we always imagine something absolute. What we have considered so far, was only relative truth: a sentence being true relative to a particular structure. So here is an important concept. There are sentences that are true in all structures. One may get the impression that such sentences are very simple and uninteresting. Also one of their names, *tautologies*, has such a connotation. But in fact, these are the sentences that we are mostly interested in, the substance of logic. In logic we are interested in absolute truth, not sentences that are true only in a particular situation, those are the subject matter of other sciences. We call the sentences true in all structures[8] *logically valid*. The term '*tautology*' is mostly used for logically valid formulas of propositional calculus.[9]

Logically valid sentences express all that logic can say about truth. If we want to know, if a sentence φ follows from another sentence ψ, we can just check, if the implication '*if* ψ*, then* φ' is logically valid. In the same way we can reduce the question whether a sentence is a consequence of a set of axioms to logical validity

[8] More precisely, true in all structures of an appropriate type.

[9] Some authors distinguish between logically valid sentences and tautologies in first-order logic and call tautologies only the sentences whose validity can be established by means of propositional logic.

of certain sentences. For example, if we want to know whether the sentence $x \cdot y = y \cdot x$ is a consequence of the axioms of groups, we just need to know whether the sentence $x \cdot y = y \cdot x$ is true in all structures that satisfy the axioms of groups, which simply means we need to know if it is true in every group. (This particular sentence expresses the commutative law and is not true in every group.)

All this looks very simple, but the conclusion that we get is extremely important: *we can define logical validity.* This is a consequence of the fact that we can define satisfaction. By saying '*define*' I mean it in the strongest sense, namely, *logical validity is a mathematical concept.* There is no other basic concept of gnoseology, the science of knowledge, that can be so unambiguously defined! At this point logic departs from philosophy and becomes a part of mathematics.

Proving Consistency and Independence by Constructing Models

One of the most important problems studied in logic is the consistency of an axiomatic system. This problem is also relevant in other theoretical fields, but the closer the field is to practice the less important it is. This is because physical reality is considered the best test of consistency. In mathematics too we can test the consistency of sentences on small finite structures, small enough to be handled by computers. Thus, for example, we can show the consistency of certain axioms of geometry using the Fano plane. But the majority of the interesting theories concern infinite structures. Therefore, we have to substitute physical reality by the world of mathematical structures.

Having a definition of satisfaction, we can formally prove that testing an axiomatic theory on structures suffices to prove its consistency. Indeed, one can prove that a sentence ϕ is either true or false in a given structure M, *but not both.* Further, one can show that if axioms are satisfied in M, then so are all their logical consequences. Therefore, if we want to prove that an axiomatic theory T is consistent, it suffices to find a structure in which all axioms of T are true, in other words, to find a model of T.

A theory without models is certainly strange. One often calls a concept void if there is no example of it. What is a void theory? It turns out that such a theory is inconsistent. So having a model and being consistent are equivalent things. This is a nontrivial fact; it is called the Completeness Theorem. I will talk about it in the next section because it concerns both semantics, the topic of this section, and the syntax of proofs, which will be the topic of the next section.

Let us see how this reduction is used in practice. The simplest situation is when the axioms are satisfied in one of the structures that we already know. If this is not the case we try to adapt or combine the existing structures to get a model that we need. In other words, we construct the model from available models. Thus, for example, we prove the consistency of the complex numbers using the *Gaussian plane*, which is the ordinary plane with axis x used for the real numbers and the axis y for the imaginary numbers. A point with coordinates (a, b) corresponds to the complex number $a + \mathrm{i}b$.

The problems arise when we need models of strong theories, in particular, set theories. Except for some very weak set theories, their models cannot be constructed from the classical standard structures such as the natural numbers and the real numbers. We will see that in these cases we have to accept the consistency as a hypothesis.

If we analyze consistency proofs more closely, we find out that we are using certain assumptions even in such simple cases as the case of the complex numbers. When proving that the theory of complex numbers is consistent our assumption is that the theory of real numbers is consistent. We may say that the latter assumption is obvious, but, strictly speaking, we are only reducing the consistency of the complex numbers to the consistency of the real numbers. Essentially in all proofs of consistency we are reducing the consistency of a theory T to the consistency of another theory S. To stress this fact we say that *T is consistent relative to S*. The consistency of S can be justified by our belief that S axiomatizes a structure that is real. Then we also believe that T is consistent and do not talk about relative consistency.

To prove that a sentence is independent of a system of axioms one can also use models. This is due to the simple relation between consistency and independence:

Proposition 3 *A sentence ϕ is not provable from a consistent set of axioms A, if and only if the set A supplemented with $\neg\phi$ is consistent.*

Consequently, it is possible to prove that ϕ is not provable from A by constructing a model in which the sentences of A are true and sentence ϕ is false. This simple proposition is the basic tool in many proofs of independence. Its power lies in the fact that it replaces a negative task—to show that no proof gives ϕ, by a positive one—to construct a model.

A nice example is Euclid's fifth postulate. Recall that this axiom is equivalent to the statement that for a line and a point not on the line there is a unique line through the point that is not incident with the given line. In Euclidean geometry, the line through the point is parallel to the given line. For centuries, people believed that this axiom is superfluous because it can be derived from the others. Only at the beginning of the 19th century did some mathematicians realize that it may not be the case. Indeed, this axiom does not follow from the others. The solution of the problem is attributed to János Bolyai, Carl Friedrich Gauss and Nikolai Ivanovich Lobachevsky. Lobachevsky and Bolyai published their works in 1829 and 1831; the only evidence about Gauss' work that we have is from his letters, but it is convincing enough. Lobachevsky and Bolyai developed what is nowadays called *hyperbolic geometry* or *Lobachevsky-Bolyai geometry*. Lobachevsky studied fairly non-trivial problems such as the volumes of polyhedra in three-dimensional hyperbolic geometry.

In logical terms, they studied the theory in which the fifth postulate is replaced by its negation. In order to prove that the fifth postulate is independent, it sufficed to show that the new theory was consistent. Lobachevsky and Bolyai considered the fact that the theory leads to meaningful results as sufficient evidence that the new theory is consistent, but they did not have a proof. It was still possible that when the theory was developed further it would run into a contradiction. A genuine proof of independence only appeared later, in 1868, when Eugenio Beltrami constructed a model of this theory. In Beltrami's model all axioms of Euclidean geometry were true, except the fifth postulate, which showed that the fifth postulate was independent. There is no doubt that the insight of Bolyai, Gauss and Lobachevsky was the major step in the solution of the problem, but the problem was solved by Beltrami.[10]

In popular expositions of the problem of Euclid's fifth postulate you can still read that "Gauss, Lobachevsky and Bolyai proved the existence of non-Euclidean geometries". Let us ponder what a proof of the existence of a concept means. In contemporary mathematics it means precisely this: to prove the existence of a structure that is a model of the concept in Zermelo-Fraenkel Set Theory. So the fact that one can develop a meaningful theory about the concept does not count as a proof; it may only be accepted as a piece of evidence supporting the conjecture. There is, however, one important exception: set theory itself. As we will learn in the next section, one cannot prove the consistency of Zermelo-Fraenkel Set Theory in itself, hence also one cannot construct a model of Zermelo-Fraenkel Set Theory in itself. So for the existence of this concept, we only have arguments based on a having "well-behaved theory" etc., arguments of the kind that we dismissed in the case of non-Euclidean geometries.

Models Are not Uniquely Determined by Theories

Assuming that structures are the main subject of our study and logic only serves to describe them, we would like logic to be able to determine each structure as much as possible. Clearly, logic cannot determine a particular structure uniquely because for a given structure there are infinitely many isomorphic ones. That is all right, we do not want to distinguish isomorphic copies of the same structure. So our concern is if one can determine a structure *up to an isomorphism*. In general, this is not possible. More precisely, one can determine only finite structures by the sentences that are true in them. For an infinite structure, there are structures that are essentially different, but which satisfy exactly the same sentences. The best way to demonstrate it is to consider the sizes of structures. A classical result of Leopold Löwenheim (1878–1957) [186] and Thoralf Skolem (1887–1963) [270] says that for an infinite structure there are structures of any infinite cardinality satisfying the same sentences. (For a more precise statement see Notes.)

Example Consider the natural numbers and the real numbers. The first structure is countable and the second is uncountable. We think of the set of natural numbers as the canonical example of a countable infinite set (in fact, '*countable*' comes from the possibility of enumerating the set by numbers). Yet, there are structures that are uncountable and they satisfy the same sentences. Similarly, the real numbers are a

[10] Gauss did important work in the study of the concept of curvature. We may thus speculate that he could have realized that curved surfaces are models of non-Euclidean geometry, but we do not have any historical evidence of that.

prototype of a higher type infinity, the continuum, but there are countable structures that are logically indistinguishable from them.

There is an even more striking example. Consider an axiomatic system for set theory, say Zermelo-Fraenkel Set Theory. Assuming it is consistent, it has a model, but then, according to Löwenheim-Skolem's theorem, it also has a countable model. It has a countable model, in spite of the fact that in this theory there are many much larger cardinalities! This looks really weird, but in fact, it is also a good example for explaining how it is possible. The crucial thing is to realize that a structure is a world that is different from ours. People living there see things from a different perspective, from a much narrower one. It is like popular explanations of higher dimensions. People living in a two-dimensional world could not escape from a circle. Watching them in a three-dimensional world, we see that it is possible to use the extra dimension to jump over the border. But only we can use the three dimensions, the rules of the game do not allow the people from the two dimensions to do so.

Turning back to the countable model of set theory, let us call it M. Take the natural numbers of M and the reals of M. As the whole model M is countable, both the natural numbers and the real numbers of M are countable sets. This seemingly contradicts the theorem of set theory saying that the two sets have different cardinalities. To resolve this apparent contradiction we have to recall the definition of what it means that two sets have the same cardinality. The definition says that they have the same cardinality, if there exists a one-to-one mapping f from one of the sets onto the other one. So let such an f be the mapping of the natural numbers of M onto the real numbers of M. To conclude that the two sets have the same cardinality from the point of view of M we would need to prove that such an f is in M, but there is no reason why it should be. So the contradiction is only apparent. Exactly like the action of jumping from the circle is not allowed in the two-dimensional world, the mapping f is not allowed in the model M.

The reason for this discrepancy is that logic is in some sense finite (the technical term for this property is *compact*). In particular, each proof is finite and therefore it cannot use more than a finite number of axioms. Hence everything that we can prove about a structure only depends on local properties. The cardinality, however, is a global property.

A Nonstandard Model of Arithmetic

Not only are there models of different cardinalities that satisfy the same sentences, but also in one given cardinality there may be different ones. This may be viewed as a drawback of logic, but also as an advantage: it gives us interesting structures. In particular such interesting structures are the *nonstandard models of arithmetic*. By arithmetic I mean a theory that partially describes the structure $(\mathbb{N}; +, \cdot, \leq)$, or more precisely, a theory T one of whose models is $(\mathbb{N}; +, \cdot, \leq)$. We call such theories *arithmetical*. An important arithmetical theory is *Peano Arithmetic* (see page 116). This is a theory based on a small set of simple axioms and an infinite set of axioms stating the principle of the mathematical induction for every formula in the language of $(\mathbb{N}; +, \cdot, \leq)$. In model theory the word '*theory*' is used for any consistent set of sentences, even if there is no effective procedure to determine if a sentence is an axiom of this theory. Such a theory is *True Arithmetic*, which is simply the set of all sentences true in $(\mathbb{N}; +, \cdot, \leq)$. We will see in Chap. 3 that nonstandard models of True Arithmetic are very useful.

To define nonstandard models, we rather define its opposite, the standard model. The standard model is simply $(\mathbb{N}; +, \cdot, \leq)$ and all models isomorphic to it. Hence, M is nonstandard, if it is not isomorphic to $(\mathbb{N}; +, \cdot, \leq)$. Thus each uncountable model of arithmetic is nonstandard, but there are also countable ones. Nonstandard models are very complex structures and they cannot be obtained by an explicit construction (except for some very weak subtheories of Peano Arithmetic). To get at least an idea of what they look like, one should look at the ordering of elements of such a model M. The model starts with a copy of the standard model. These are the elements that can be denoted by numerals; they are called *standard numbers* or *finite numbers*. Since M is not standard, there must be other elements. They are all after standards numbers and they are called *nonstandard* or *infinite*. Clearly, there is no largest nonstandard number, but also there is no smallest nonstandard number. In fact, for a nonstandard number α, the number $\alpha - 1$, and the integer parts of $\alpha/2$, $\sqrt{\alpha}$, etc. are also nonstandard. Note that the fact that there is no least nonstandard number does not contradict the least number principle because the set of nonstandard numbers is not definable in M.

Notes

1. *The definition of satisfaction.* Assume that a finite list of nonlogical symbol is given. To simplify the definition I will only use relation symbols R. Further, I will assume that the logical symbols are only $\wedge$ (conjunction), $\neg$ (negation), and $\exists$ (existential quantifier). Let $\mathcal{L}$ be such a language. Given a relation symbol R from $\mathcal{L}$ and a structure M for $\mathcal{L}$, I will denote by R^M the relation of M that is the intended interpretation of the relation R in M.

   For a formula ϕ with free variables $x_1, \dots, x_n$ and elements $a_1, \dots, a_n$ from M, we want to define the relation '*ϕ is satisfied by elements $a_1, \dots, a_n$ in M*'. To simplify the definition I will further use the standard notation $M \models \phi[a_1, \dots, a_n]$ to denote this relation of satisfaction. The definition goes by induction on the complexity of the formula ϕ:

   a. for a k-ary relation symbol $R(x_1, \dots, x_k)$, $M \models R[a_1, \dots, a_k]$ if the elements $a_1, \dots, a_k$ are in the relation R^M;
   b. for formulas ϕ and ψ, $M \models (\phi \wedge \psi)[a_1, \dots, a_n]$ if $M \models \phi[a_1, \dots, a_n]$ and $M \models \psi[a_1, \dots, a_n]$;
   c. for a formula ϕ, $M \models \neg\phi[a_1, \dots, a_n]$ if it is not true that $M \models \phi[a_1, \dots, a_n]$;
   d. for a formula ϕ, and a variable x_1, $M \models \exists x_1 \phi[a_2, \dots, a_n]$ if there exists an element a_1 in M such that $M \models \phi[a_1, a_2, \dots, a_n]$.

Since this is a very important definition, I will describe in more detail how it is formalized in set theory. Let M be a fixed model. Let $\mathcal{F}$ be all first-order formulas in language $\mathcal{L}$ and let A be all finite sequences of elements of M. Formulas are formalized by finite strings of symbols from a finite alphabet. Let $\mathcal{F}_i$ denote the subset of $\mathcal{F}$ consisting of all formulas of logical complexity i. Thus $\mathcal{F}_0$ are atomic formulas, $\mathcal{F}_{i+1}$ are all formulas from $\mathcal{F}_i$, plus those that are obtained from them using a connective or a quantifier. We have $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \cdots$, and $\mathcal{F}$ is the union of the sets $\mathcal{F}_i$, $i = 0, 1, \ldots$. Then we define the relation $\models$ between $\mathcal{F}_i$ and A by recursion on i. Condition 1. defines the base case. Conditions 2.–4. define how to extend the relation $\models$ from $\mathcal{F}_i$ to $\mathcal{F}_{i+1}$. Thus we obtain a sequence of relations, where the first relation is defined explicitly and each succeeding one is defined explicitly from the previous one. Finally, the relation $\models$ is the union of all these partial relations.

Note that the definition of truth of sentences is a special case of this definition: a sentence ϕ is true in M, if it is satisfied by the empty string in M.

The definition of satisfaction for higher order structures and languages is analogous.

2. *The Löwenheim–Skolem Theorem.*

Theorem 2 *If a theory T has an infinite model, then it has models of arbitrary infinite cardinalities.*

The theorem is a consequence of the proof of the completeness theorem, which we will present in the next section. Here I will only sketch a proof of a weaker theorem:

If T has an infinite model, then it has a countable model.

Furthermore I will only consider the special case where the theory has a single axiom of the form $\forall x \exists y\ \phi(x, y)$, where $\phi(x, y)$ is quantifier-free. Assume the language of the theory does not contain function symbols. Let M be an infinite model of this theory. Let us define a function f on the universe of M by choosing, for every x in M, an element $f(x)$ such that $\phi(x, f(x))$ holds in the model. (f is called the *Skolem function* for $\forall x \exists y\ \phi(x, y)$.) Pick an element a of the model and take M' to be the submodel of M with universe $\{f^n(a);\ n = 0, 1, 2, \ldots\}$, where f^n denotes n-times iterated f. Since ϕ is quantifier free, $\phi(f^n(a), f^{n+1}(a))$ is true also in the submodel M', thus M' satisfies the axiom. M' is either countable infinite or finite. If it is finite, repeat this process with another element, etc.

If the language does contain function symbols, we have to take the universe of M' to be all elements generated from a by the functions of M and f.

The above proof can be explained as follows (the same idea is used to prove the theorem in general). First we replaced the theory by a universal theory, namely, the theory axiomatized by $\forall x\ \phi(x, f(x))$. Then we have taken a substructure of M generated from a single element. A substructure satisfies all universal sentences that the structure does, hence it is also a model of the original theory. For more detail, see also *Compactness* on page 115.

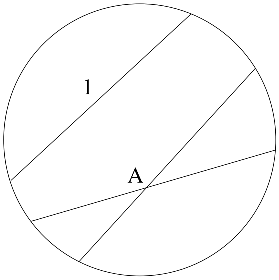


Fig. 2.1 The Beltrami-Klein model of the hyperbolic plane

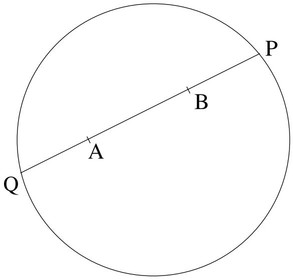


Fig. 2.2 The hyperbolic length of the segment AB is $\frac{1}{2}\log\frac{|AP|\cdot|BQ|}{|AQ|\cdot|BP|}$, where $|\cdots|$ denote the Euclidean lengths of the segments

3. *Models of planar hyperbolic geometry.* In hyperbolic geometry the fifth postulate is replaced by

   > *For every line l and every point A not on l, there are at least two different lines going though A that are not incident with l.*

   The first model of non-Euclidean plane geometry found by Beltrami was on surfaces of negative curvature. Later he realized that one can project it on a disc in a Euclidean plane. This model was popularized by Felix Klein, thus it is often called the Beltrami-Klein model. The points of this model are points inside of the circle (the points on the circle are excluded). The lines of the model consist of the inner points of segments whose endpoints are on the circle, see Fig. 2.1. The verification of the axioms of incidence, including the axiom that replaces the fifth postulate, and the axioms about the relation '*between*' is very easy because both relations are the same as in the Euclidean plane from which is the model constructed.

   The tricky part is to define the relation of congruence between segments and to prove that the corresponding axioms are true. To this end, one defines the hyperbolic length of a segment, see Fig. 2.2. Then two segments are congruent (have the same length) in the new sense if the real numbers assigned to them are the same.

   A different model was discovered by Poincaré. The points of the model are again the inner points of a circle. Lines of the model are arcs (sections of circles) that start and end on the circle and that are perpendicular to the circle at both sides. Moreover, every diameter is a line too. The advantage of this model is that

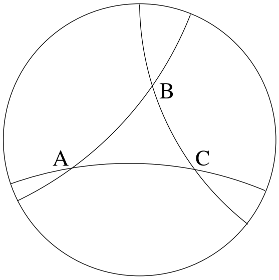


Fig. 2.3 Poincaré's model

the angles in the model are the true angles. Thus you can clearly see that the sum of the angles of the triangle ABC in Fig. 2.3 is strictly less than 180°.

Note that both models are constructed from the Euclidean plane. Thus this proves *relative consistency*: if the axioms of the Euclidean plane are consistent, then so are the axioms of the hyperbolic plane.

4. *Nonstandard models of* $(\mathbb{N}; +, \leq)$ *and* $(\mathbb{R}; +, \cdot, \leq)$. In contrast to the arithmetic of the natural numbers with addition and multiplication, these two structure determine fairly weak theories. Therefore, it is possible to construct explicitly nonstandard models of them.

   Here is a nonstandard model M of the addition of integers, the theory of $(\mathbb{N}; +, \leq)$. The universe consists of pairs (q, n), with n an integer and q a nonnegative rational number; however we exclude pairs $(0, n)$ for $n < 0$. The ordering of M is the lexicographic ordering determined by the orderings of integers and rational numbers. The addition is defined componentwise, $(q, n) +_M (r, m) = (q + r, n + m)$.

   We know that Peano Arithmetic certainly does not have such a simple model. One result that proves this is a theorem by S. Tennenbaum. This theorem gives a lower bound on the complexity of countable nonstandard models of Peano Arithmetic: they cannot be constructed using computable relations and operations.

   The theory of $(\mathbb{R}; +, \cdot, \leq)$ is called the *Theory of real closed fields* because it is axiomatized by the axioms of ordered fields and the axiom schema that says that every polynomial of an odd degree has a root. To get a countable model of this theory, one can simply take the submodel of this structure determined by algebraic numbers (numbers that are solutions of algebraic equations with integer coefficients).

   Note, however, that all this is due to the restricted language. In the structure $(\mathbb{R}; +, \cdot, \leq)$ it is impossible to define integers. If we enrich the structure by predicates or functions that allow us to define integers, axiomatization by a decidable set of axioms becomes impossible, as well as the construction of computable nonstandard models.

5. *Free algebras.* Let a theory T be given. We would like to have a most general model of T, so that we can study what is provable in T and what is not. In general there is no canonical way to assign any such model to T, but in some cases it is possible. The problem is that while in a theory T there may be undecidable sentences, in a model, for every ϕ, we must have either ϕ or $\neg\phi$; but there is

no rule that would tell us which of the two is better. The most important class of theories that do have such canonical models are equational theories. These are theories that are given by a set of elementary positive statements—equations. Then if we are to decide which of the two independent sentences $s = t$ or $\neg s = t$ should hold, we always take the second. Thus we obtain an algebra that has in some sense the least possible number of dependences among its elements.

This is only a rough description. To make it work one has to start with a countable set of elements, called *generators*. The resulting algebra is called the *free algebra* (with countably many generators). Then an equation $t = s$ with variables $x_1, \dots, x_n$ is derivable in the equational theory if and only if it is satisfied by n distinct generators of the free algebra.

6. *Logics without semantics and logics without syntax.* I have presented the standard approach that assumes that there is a world of structures (or models) and logic is a means of talking about them. Hence a logical system that does not have semantics is considered meaningless. Nevertheless, other philosophical directions in the foundations of mathematics refute this basic dogma. They say that the only real entities are proofs. Then having semantics is not considered important at all. There are logical systems that have a nice syntax, which means that they have certain plausible properties, but for which finding semantics was a problem (see, for example, Church's λ-*calculus*, page 146).

   On the other hand, there are logics that miss an important part of syntax. Any system called logic must have formulas to express statements, but some systems do not have the concept of a proof, or if they have a concept of a proof, there is no completeness result saying that all logically valid sentences are provable. This concerns all higher order logics starting with second-order logic. There are also logics that use infinitely long formulas and infinitely long proofs. This can be accepted as syntax, but it is not a kind of syntax that we can use for practical purposes.

2.3 Proofs

The definition of truth determines the set of logically valid sentences, but it does not give us a way to determine which are these sentences. Applying the definition would mean testing the truth on each structure. There is no effective procedure for that even for a single structure. This does not mean that the definition is completely useless for that purpose. For a particular sentence, we might be able to prove, using mathematical considerations, that it is true in all structures. But why should we look for a proof of the fact that a sentence φ is true in all structures, if we can prove φ itself?

This suggests that there is another way to define when a sentences is true—by a proof. The concept of a proof is an old one. It has been used mainly in two fields: mathematics and law. The meaning of the concept is a collection of pieces of evidence that is able to persuade any person about the truth of a sentence in question. The pieces of evidence alone do not suffice, they have to be arranged in such a way that they fit together. In criminal investigations and lawsuits the evidence is the facts related to the case and testimonies of reliable witnesses. In mathematics the evidence is axioms, previously established theorems, and computations. The pieces of evidence are just some sentences, thus the essence of the concept of a proof hinges on how they are connected.

In lawsuits one should prove the accusation '*beyond a reasonable doubt*'. Clearly, it is unreasonable to ask for proof without any doubt, as nobody would be convicted. The problem is in that any presented evidence and testimony can be questioned as nothing in the world is one hundred percent sure. To support the questioned evidence new proofs may be required and so on resulting in a never-ending process. But even if all the facts and testimonies are accepted as unquestionable, there are still a lot of hidden assumptions that one has to use. People who try to simulate human reasoning on computers know that even in reasoning about simple situations we unconsciously use hundreds of assumptions.

In mathematics the first proofs also were very incomplete. But already in Euclid's *Elements* the proofs are essentially the same as in contemporary mathematics (except that they sometimes use assumptions not stated explicitly as axioms). The concept of a proof, however, was not defined back then. Aristotle studied *syllogisms*, which are some rules for propositional logic, but he did not have a complete system. In the 17th century the German mathematician and philosopher Gottfried Wilhelm Leibniz (1646–1716) proposed the idea of designing a calculus for logic in which one could present logical reasoning and check its correctness in the same way as we calculate in algebra. It was, however, only two centuries later, when his vision was realized. The first complete system for first-order logic was constructed by Frege in his *Begriffsschrift* in 1879 [77].[12] Peano presented another system in his *Principles of Arithmetic* in 1889 [216]. A.N. Whitehead and Russell published their fundamental work *Principia Mathematica* on the foundations of mathematics in three volumes in 1910, 1912 and 1913 [313–315]. Their formal system for proofs in first-order logic became the prototype for many systems that appeared later.

Gottlob Frege[11]

We say that the concept of a proof is now *formal*, meaning that it can be treated as a rigorous mathematical concept. Note that formalization of the concept of a proof was only needed for understanding the foundations of mathematics. It is remarkable that every mathematician has a perfect sense for what constitutes a proof and what does not. So to say, mathematicians learn the rules of the game by playing it.

Claiming that we have a formal definition of a proof we should give such a definition quite formally. I will do it in the sequel; here I only give a short description how it can be done. Firstly, we take some simple sentences which are obviously tautologies as *logical axioms*. A typical one is the *law of excluded middle*, which is φ *or not* φ; we postulate it for every formula φ. Then we need *logical rules* that enable

[11] This media file is in the public domain in the United States.

[12] The title is translated as *Concept Script*.

us to derive tautologies that are not axioms. The rule *modus ponens* used already by Aristotle is a case in point. This rule says:

Suppose 'φ' and 'if φ, then ψ' *has been established. Then we can derive 'ψ'.*

A proof is constructed by successively deriving sentences. This means that at each step we either simply write down an axiom or derive a sentence from previously obtained ones using a logical rule. Note that at each step we have several options what to do next. The sentence that we want to prove gives us only hints, but there is no general strategy. The daily experience of mathematicians is that sometimes it is extremely hard to find a proof of a given sentence.

Once we list all symbols used in the language, all axioms and all rules, the concept becomes completely formal. The proof is a sequence of symbols satisfying certain syntactical rules. It is simpler to describe the syntax of a proof than, say, the syntax of an English sentence, even not counting the complexity of the English vocabulary. All logicians agree that the concept of a proof can be made completely formal. However, there are some mathematicians who doubt that all actual proofs produced by mathematicians can be converted into such formal proofs. The purpose of a real mathematical proof is not to present a sequence of symbols that can be mechanically checked, but to convey the idea that leads to a proof and persuade the reader that the idea can be realized. Thus for instance, if the proof splits into considering several similar cases, a proof for only one is given and the others are left to the reader to prove, or it is only stated how the proof for the first case can be modified to work for the other cases, etc. Furthermore the steps in such a proof are not elementary steps as in a formal proof, they are more like small jumps. Depending on how big these jumps are, the proof requires more or less effort and ingenuity on the part of the reader. Careful checking of a ten-page article may take several days, but there are some that need many more. If we have problems with short proofs, what would happen if we tried to convert a really long one into a formal proof?

Can All Mathematical Proofs Be Turned into Formal Proofs?

I contend that, if needed, we would be able to formalize any proof with only a moderate amount of work. I think that our experience with computers demonstrates it sufficiently clearly. That experience shows that any algorithm can be formalized. Before the advent of computers nobody cared to write down algorithms formally. Nowadays, when we need to communicate a lot of algorithms to computers, there are algorithms of extreme complexity written formally. This was certainly assisted by the use of high level programming languages. While people write programs routinely, writing a mathematical proof in a formal system so that a computer can check it is only done by researchers in the field of proof checking and a few mathematicians. Proof checkers and automated theorem provers had been written almost as soon as computers became available. Already in the late 1960s, Hao Wang (1921–1995) wrote a program that proved all the approximately 350 theorems of *Principia Mathematica*, (see [303]). Since then the power of computers has increased dramatically and many proof checkers and programs for automated theorem proving have been written. The interest of working mathematicians in these programs is growing and the demand for such programs in industry is significant. It is just a matter of time when applications of formal logic become an important part of the computer industry. Then writing formal proofs will be as routine as writing programs.

The book *Proofs and Refutations, The Logic of Mathematical Discovery* [178] by Imre Lakatos is an interesting treatise on mathematical proofs. The leading idea is that mathematical theories can turn out to be wrong in the same way as it happens with physical theories. A physical theory, however nice it is, has to be refuted, if there is an experiment which is in contradiction with the prediction of the theory. In physics it is not a theoretical possibility, but rather a typical process—theories are proposed and refuted only to be replaced by more accurate ones. This goes on and on. Lakatos's claim is that the same happens in mathematics. As a case example, he considers proofs of the well-known Euler formula which asserts that for each polyhedron the number of vertices plus the number of faces equals to the number of edges plus 2, which is expressed as

$$v+f=e+2.$$

He shows how proofs of this theorem were found and then refuted by counterexamples, then fixed again and so on. He hints that this is a never-ending process. I do not think that by claiming that proofs are never quite correct he wanted to harm mathematics. Apparently, he was inspired by Karl Popper's *Conjectures and Refutations* whose main thesis is that a scientific theory must be refutable at least in principle. Ideologies claiming their infallibility are good only for manipulating people. This is a great idea of a great philosopher, but should it be applied to mathematics too? There is a fundamental difference between mathematical results and physical theories. When a physical theory fails, it means that the given equations are not applicable to the reality they should have represented. It is not that the theory as a mathematical work is wrong. Mathematical results, though inspired by our physical experience, are derived without referring to it; they may have one or more useful applications, but some do not have any, and this is not considered as a failure.

Lakatos talks on refutations of proofs, but it is not the concept of a proof which is not rigorous, the problem is in the definitions of the concepts used in the proofs. The troubles are caused mainly by the definition of polyhedra. There are several classes of three-dimensional shapes that we might be willing to call polyhedra. If we take the most restrictive definition, a simple proof works well. For more general forms, the proof fails and we may need a different proof or the theorem is not true at all. Proofs are refuted because some tacit assumptions turn out to be wrong. It is not logic that fails.

Some mathematicians may still doubt that it is possible to spell out all the assumptions in deeper mathematical proofs. This was indeed the case until the 19th century. By the end of that century all mathematical concepts had been precisely defined and that became a standard for the future. The concepts introduced into mathematics later were always constructed rigorously. Furthermore also the concept of a proof itself became a part of mathematics. I admit that some proofs would need a considerable amount of work to be turned into formal proofs, but I am sure it would be less work (and, perhaps, more interesting) than writing an average size computer program. According to current estimates, based on the experience with writing formal proofs for computer checkers, the size of a formal proof is on the average about three to four times bigger than a plain "mathematical" proof. This may seem too much for being accepted by mathematicians. But a four-times longer text does not mean that one needs four times more *time* to write it. Every mathematician knows that writing up a mathematical result is always the easier part of the job, usually much easier than discovering the result. There is another argument that suggests that mathematicians might be willing to devote a little more time to writing their proofs if they could gain something. It is the experience with typesetting mathematical papers and, in particular, mathematical texts. Nowadays mathematics (including books like this) is typeset in the system TEX (or various versions of it) designed by the American mathematician and computer scientist Donald Knuth. If you compare the difficulty of writing a formula in TEX with writing it by hand you probably get a similar factor (about 3:1) for the ratio between formal and informal proofs. Yet most mathematicians prefer typing their papers in TEX themselves.

In 1892, Peano started an ambitious project called *Formulario Mathematico*, [217]. The aim of the project was to write a collection of all known mathematical theorems in a formal logical language. He expected that the advantage of presenting mathematics in this way would be brevity, precision and uniformity. The succinctness of logical formulas would enable him to cover a large amount of material, which otherwise would be impossible. The project was finished in 1908, but it was rather a failure—nobody was using it, except for Peano. Although he described the main theorems also in words, it was very difficult to read it because most of the text were just formulas containing a lot of new symbols. Nevertheless, some symbols introduced by Peano have been accepted and have been used since then; for instance the set-theoretical symbols $\in, \subset, \cup, \cap$.

Peano's idea was that mathematics would be taught using *Formulario.* He gave such courses of calculus, but the students were desperate. His great mistake was that he did not realize that mathematical texts have a twofold role. The first one is to verify the truth of the stated theorems. Therefore we need precise language, precise definitions and precise proofs. It is true that the highest level of precision is only attainable by formal systems, but they do not have to be based solely on symbols. The second role, the one that Peano neglected, is to communicate ideas to other people. It is possible to memorize a formal proof, but it is useless. Mathematicians need to understand the proof. They need to make a "mental picture" of the proof and put it into context of related results etc. Again, there is a parallel with computer programs. A computer only needs a formally precise program, but studying an uncommented complicated program is a programmer's nightmare.

One reason some mathematicians do not believe that proofs can be written formally is that introductory textbooks most often use formal systems for first-order logic that are simple to describe, but difficult to use. These systems were developed with the aim to have as simple definitions of the formal systems as possible. There are, however, other systems that are aimed at practical use. Proofs in these systems are indeed very close to the way mathematicians argue in their proofs, therefore they are called *natural deduction* systems. I will describe such a system in Notes; below I will only give an example of a proof in this system.

Example Let us compare a mathematical proof and a fully formalized proof in a natural deduction system. We want to prove the tautology:

If it is not true that A is true and B false, then either A is false or B is true.

Here is how a mathematician would prove it. (Mathematicians would do it only for didactic reasons because the tautology is "obviously true".) The numbers in parentheses are added for the sake of comparing the mathematical proof with the formal proof below.

Suppose that it is not true that A is true and B is false (1.). *Consider two cases.*

Case 1, *A is true* (2.). *Arguing by contradiction, suppose that B is false* (3.). *Then A is true and B is false, which contradicts to our initial assumption. Hence B must be true* (5.). *This implies that either A is false or B is true* (6.).

Case 2, *A is false* (7.). *Then we have again that either A is false or B is true* (8.).

Since A is either true or false (9.) *and in both cases either A is false or B is true, the latter fact is always true* (10.).

Since we have derived that either A is false or B is true from the assumption that it is not true that A is true and B is false, we have proved the implication (11.).

In logic the tautology is expressed by

$$\neg(A \wedge \neg B) \rightarrow (\neg A \vee B).$$

Here is a formal proof in the natural deduction system presented on page 113.

1. $\neg(A \wedge \neg B)$ [assumption]
2. $\quad A$ [assumption]
3. $\quad\quad \neg B$ [assumption]
4. $\quad\quad\quad A \wedge \neg B$ [from 2. and 3. by rule 1]
5. $\quad\quad B$ [from 1. and 4. by rule 8]
6. $\quad\quad \neg A \vee B$ [from 5. by rule 3]
7. $\quad \neg A$ [assumption]
8. $\quad\quad \neg A \vee B$ [from 7. by rule 3]
9. $\quad A \vee \neg A$ [axiom]
10. $\quad \neg A \vee B$ [from 9., 6. and 8. by rule 4]
11. $\neg(A \wedge \neg B) \rightarrow (\neg A \vee B)$ [from 1. and 10. by rule 5]

The main problem of proof checkers (and automated theorem provers as well) is their lack of the basic knowledge that all mathematicians have. For most results, should they be represented formally, one needs to introduce a lot of concepts and prove a lot of elementary theorems. This depends on the fields of mathematics. In set theory it is not such a problem, since what we start with are axioms of set theory. That is why Wang was able to prove all theorems of *Principia Mathematica* [303]. In other fields one has to go a very long way from the axioms of set theory. Thus some proof checkers are only built for a single mathematical field. To overcome this problem we need to create a large "mathematical library" of basic concepts and theorems, so that then authors can use some concepts and refer to some theorems instead of proving everything from scratch.

The second big problem is the lack of a sufficiently strong versatile automated theorem prover. When writing a proof, a mathematician leaves a lot of simple logical deductions to the reader (sometimes they are not that simple, sometimes they are gaps that cannot be fixed). It would be very boring, also for the reader, if every simple detail were spelled out. So this has to be done by computer and eventually will, but it will take time.

Now that I have spoken so much about the exactness of proofs the reader may have become worried that I want to hide something more fundamental, without which the above question does not have a proper meaning, namely:

Is the Concept of Logically Derivable Sentences Uniquely Determined?

One can propose a large number of different logical calculi, thus the question is whether they differ also in the theorems that they are able to prove. This is a really fundamental question, it is the question about what logic is. Recall that in the previous section we gave a semantical definition of true sentences. Our point of view is that structures are primary and logic is a means to describe them. Therefore, our aim should be to show that true sentences are exactly those that are provable, which will automatically solve the problem of the uniqueness. Showing that the semantical definition of logically valid sentences is the same as the syntactical one entails two things:

1. *one can only prove logically valid sentences, and*
2. *every logically valid sentence can be proved.*

The first property is called the *soundness* of the logical calculus and it is fairly easy to prove it. One shows that the logical axioms are true and that logical rules preserve truth, and then we apply the principle of mathematical induction to prove that all derivable sentences are true. Checking axioms and rules is easy, as they have a very simple structure. Let us look, for instance, at *modus ponens* (see page 94). We want to show the following: if φ is true and φ *implies* ψ is true, then also ψ is true. But that is obvious, because if φ *implies* ψ is true and we know that φ is true, then also ψ must be true.

It is as if I heard you saying: '*Wait a moment, what kind of proof is this! You are proving the soundness of modus ponens by using modus ponens!*'. Indeed, this is what we do. But notice that this is exactly the same situation as it was in the definition of satisfaction. We are just translating the logical axioms and rules from the metalanguage into a formal system which is the object of our study.

Also here it helps to imagine that the formalization is intended for a computer. Suppose that we have written a program for automated theorem proving and now we want to check that it is correct. Namely, we want to prove that it can only prove logically valid sentences. Then we would argue in exactly the same way as sketched above. We would say that it can only start with logical axioms, which are obviously logically valid, and derive sentences by rules that are obviously logically sound.

Note that we do use a mathematical argument in this proof, although a very easy one—we are using mathematical induction to prove that all derived sentences are logically valid.

Those who doubted the usefulness of logic before starting to read this book may view this as a confirmation of their opinion. Indeed, it looks like all logicians do is just rewriting obvious things in an obscure formalism, but we will shortly see that this is not the case.

Let us now turn to the second property. It concerns the question about the *completeness* of logical calculi: are our logical calculi complete in the sense that one can derive all true sentences from them? Here we meet Kurt Gödel (1906–1978) for the first time with his first fundamental result:

The Completeness Theorem *The calculi for first-order logic are complete.*

In his doctoral dissertation from 1929 (published in a journal in 1930 [95]), Gödel proved the theorem for the calculus presented in *Principia Mathematica*. But the choice of the particular calculus was not important because all proposed calculi are equivalent.

The soundness and completeness means that the semantical definition of logical validity coincides with the syntactical definition. This has one profound consequence. Since proofs are finite entities that we can explicitly construct, logical validity is a concept that is fully accessible to us.

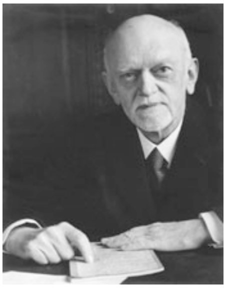
Kurt Gödel
Courtesy of Kurt Gödel Society, Vienna

Maybe completeness does not come as a surprise to you and you expect a similar kind of a “silly argument” that I used to prove soundness, but completeness is far less obvious and requires a nontrivial proof. The reason is that the two properties, soundness and completeness, are not treated in the same way. The soundness is clearly a necessary condition that we require from any logical calculus, and that we had better put it in the definition of a logical calculus. Of course, we would like to have both properties, but we would not sacrifice soundness for the sake of completeness. This is not a purely academic question, as in general completeness does not hold always. It holds for the most common logic, first-order logic, but there is no way to extend it to stronger logics which contain set theoretical notions. I will give an informal sketch of the proof of the Completeness Theorem in the sequel.

At this point it is difficult to fully appreciate the fact that it is possible to completely characterize the true sentences by the syntactical concept of a proof (expressed briefly, that we have soundness and completeness). It is a great achievement that the question of what logic is has been solved once and for all. We will see in the next section that, in spite of similarities between the concepts of proofs and computations, we are not able to prove that the commonly accepted definition of computations is the right one.

The First Incompleteness Theorem

We know what logic is, we know what proofs are, and we can prove every logically valid sentence. So why are we not yet content? The problem lies in the distinction between logical validity and mathematical truth. First-order logic, which works so perfectly, tells us what is true in *all* first-order structures. Though this is an important part of mathematical truth, there are other things in mathematics. The most interesting results in mathematics concern single structures. For example, in number theory, we study the structure of natural numbers. When proving theorems about numbers we, of course, use logic, but we also need specific properties of the structure of numbers that are not of a pure logical nature. On top of logic, we also need axioms about numbers. We need axioms in geometry, we need axioms about the real numbers, the real valued functions, and so on. By accepting set theoretical foundations we reduce the problem of finding axioms for particular structures to finding axioms for sets, but as we have seen, it is not a simple task to find axioms for sets. Thus we may give up set theory and try to find the axioms at least for the theory of numbers. When we restrict the project of finding a complete axiomatization only to the natural numbers, it does not look too ambitious. After all, natural numbers are the most familiar entities that we find in mathematics and they are so concrete. Furthermore, we know the basic principle that governs the natural numbers: the principle of mathematical induction. Still, it turns out that it is impossible to find such an axiomatization. Let me stress that the problem is not caused by an extreme complexity or difficulty, it is simply impossible.



David Hilbert
Courtesy of
Mathematisches
Forschungsinstitut
Oberwolfach

This, certainly, needs more explanation, but let me make a brief digression into history. The aim of *Hilbert's Program* was to find axioms for all the basic mathematical structures and proofs of the consistency of these axiomatic systems. The program was presented in Hilbert's address at the Third International Congress of Mathematicians, held in Heidelberg in 1904. Rudiments of his program are also the content of the second problem of the famous twenty three open problems presented on the previous International Congress of Mathematicians, held in Paris in 1900. The problem asked for proving the consistency of an axiomatic system for the real numbers.[13] A prototype of axiomatizations that Hilbert asked for was his axiomatization of geometry. Proofs of consistency were also quite common in geometry, where the consistency was shown by constructing a model for a given set of axioms. If successful, the program would guarantee that whatever meaning mathematical results have, we would

[13] His goal was not to find axioms for the structure of real numbers with only operations $+$ and $\times$. He wanted to axiomatize a richer structure in which at least a basic mathematical analysis could be done.

at least know that proving theorems is not a completely futile activity. In a consistent theory the provable theorems are separated from refutable ones, so by proving a theorem we get a piece of information. Having a complete list of axioms for a given mathematical structure would, furthermore, give us the confidence that at least potentially we can prove every true theorem. This would not solve the problems of foundations completely, as there still would remain questions such as what mathematics has to do with reality, where do the structures come from, etc. But, in some sense, it would leave the problems to philosophers—the remaining problems would not be mathematical problems.

Hilbert was very enthusiastic about this project and several mathematicians, in particular the Swiss logician and philosopher Paul Bernays (1888–1977) joined him. They founded a new field of logic, *proof theory*,[14] the main aim of which was to achieve the goals of Hilbert's Program. But after initial success came a blow. In 1930 at a conference in Köningsberg Gödel presented his incompleteness theorem, which we call the *First Incompleteness Theorem.* This theorem showed that it is impossible to achieve the goal of Hilbert's Program because every theory that formalizes a sufficiently large part of mathematics is necessarily incomplete. The Second Incompleteness Theorem had even more profound impact on Hilbert's Program, but let me first explain the simpler of the two theorems.

In order to state the First Incompleteness Theorem more formally, we need to understand the concepts involved. Recall that a formal theory is a theory axiomatized by a decidable set of axioms. The decidability means that using an algorithm we can decide whether or not a given sentence belongs to the set of axioms. In practical theories, the decision procedure is very simple because a typical theory has a finite list of axioms and, possibly, a finite list of axiom schemata.

The second thing that needs specification is what part of mathematics should the theory axiomatize. In his paper Gödel used the Theory of Types of Whitehead's and Russell's *Principia Mathematica* (as the title of his paper suggests "*On formally undecidable sentences of Principia Mathematica and related systems I*") [96]. Focusing on *Principia Mathematica* was a natural choice, as this was the most important of the formal theories that aimed at formalizing the whole of mathematics. Gödel showed that not only this theory is incomplete, but so is any extension.

Requiring the entire Theory of Types to be in the theories about which the Incompleteness Theorem speaks is a fairly strong assumption. Weakening this assumption is an important problem because one may hope to be able to completely formalize at least some parts of mathematics, and it was known that in some cases it was possible (for example, elementary geometry). Gödel was aware of the fact that his theorem can be proved for other theories and he explicitly mentioned Zermelo-Fraenkel Set Theory, von Neumann's theory and a version of Peano Arithmetic. Analyzing Gödel's proof, it is not difficult to see that the proof only needs some *finite* mathematics to be a part of the theory. The theory should be able to prove some basic facts about finite structures—finite sets, or finite strings, or natural numbers. Of the

[14] *Die Beweistheorie* in German.

three mentioned concepts, the last one is part of the most traditional mathematics. Therefore, the modern presentation of the theorem assumes that the theory formalizes the basic concepts of arithmetic and certain basic theorems about arithmetic are provable in it. One should, however, keep in mind that using arithmetic in the statement is only an elegant way of presenting the theorem and equivalent theorems can be stated using finite sets, or finite strings.

Another important condition is the consistency of the theory. If a theory is inconsistent, then it is complete in a really bad way—everything is provable. When talking about complete theories we usually have in mind those that are also consistent. For such theories, exactly one of the sentences ϕ or $\neg\phi$ is provable for every ϕ, whereas in inconsistent theories both are provable. Although we are only interested in consistent theories, when stating a general theorem we have to take care of the undesirable case of inconsistent theories. Gödel used a stronger assumption, called ω-consistency, which was later weakened to mere consistency of the theory by J.B. Rosser.

Now we know all we need to state the theorem.

The First Incompleteness Theorem *Any consistent formal theory T that is able to formalize a certain part of arithmetic is incomplete. More precisely, there is an arithmetical sentence ϕ such that neither ϕ nor its negation $\neg\phi$ is provable in T.*

The remarkable fact is that the unprovable sentences concern finite structures and numbers—the concepts that are at the heart of mathematics, the entities that most mathematicians view as real (real at least in the sense that there should be no ambiguity what is true and what is false).

Higher Order Logics and Theories

What we have considered so far is first-order logic. 'First-order' because we have variables and quantifiers only for the lowest order objects, relations and functions are fixed. What happens if we introduce variables for relations and allow quantifying them? As far as the language and its interpretation are concerned, there is no problem; we can expand the language in this way. Thus we obtain second-order languages—the languages for second-order structures. Second-order languages enable us to define interesting things; for example, we may define that 'two objects are the same if every property that one has, the other also has'. The next step is to define the satisfaction of second-order formulas in second-order structures. This also causes no problems; the definition of satisfaction for first-order formulas extends naturally to the second order. Having the definition of satisfaction, we can define logically valid second-order sentences—the sentences satisfied in all second order structures.

In order to define second-order logic, it still necessary to find axioms and rules of this logic. But here we run into an essential problem. The incompleteness phenomenon concerns also this logic: *there is no formal system by which the set of* *logically valid sentences could be defined.* In other words, we may propose some system of logical axioms and rules, but if such a system is sound, then it is incomplete. Thus what is called '*second-order logic*' is only the set of logically valid second-order sentences; there is no second-order calculus.

This paradigm can be used to define logics of any order. But all these logics suffer the same problem as second-order logic because they contain it as a subsystem. One can also take a system that contains all these finite order logics, which looks like the ultimate system, the most general logic of all. For such a logic, Whitehead and Russell introduced the Theory of Types. But their system is incomplete, and, due to Gödel, we know that it is incomplete for an essential reason.

Therefore, we should not classify these higher order logics as logics at all. The possibility to characterize logically valid sentences using a formal system is an essential property of logics. The most natural place to draw the line between logics and set theories is between first-order logic and second-order logic. Note that already in the third order theory of arithmetic one can express some important sentences and concepts of set theory such as the Axiom of Choice, the Continuum Hypothesis, the determinacy of infinite games, etc.; in weaker forms they can also be defined in the second-order theory of arithmetic.

Higher order logics were intended to be the base logics of theories about higher order structures. Since higher order logics are not axiomatizable, first-order logic is also used as the base logic for higher order structures. For example, there is a very natural (but incomplete) axiom system for the second-order structure of natural numbers, which is the standard structure of natural numbers extended by subsets. This theory, called *Second Order Arithmetic*, has two sorts of objects, numbers and sets, an both are treated as first order elements (see page 295). This is the same as in set theories, where sets of any type are treated as elements and the membership relation $\in$ is a binary relation defined on these elements.

The realization that there is only one logic, namely, first-order logic, was an important step in the development of mathematical logic. This idea is due to Skolem [271].

The Second Incompleteness Theorem

The next natural question is: what is the meaning of the independent sentence constructed in the proof of the First Incompleteness Theorem? Gödel found the answer soon after he proved the First Incompleteness Theorem (in the same year 1930). He proved a strengthening of the First Incompleteness Theorem, which is called the *Second Incompleteness Theorem*, that specifies the nature of the incompleteness. According to this theorem the unprovable sentence expresses the formal consistency of the theory.

The Second Incompleteness Theorem *If T is a consistent formal theory which is able to formalize a certain part of arithmetic, then T does not prove its own consistency.*

John von Neumann, who learned Gödel's First Incompleteness Theorem in Köningsberg, discovered the Second Incompleteness Theorem independently too. The letter in which he announced his result to Gödel arrived just three days after Gödel sent his paper [96] with both theorems to the publisher.[15]

The Second Incompleteness Theorem is not a strengthening of the First. There are consistent theories which prove sentences expressing their *inconsistency*. For such a theory, the Second Incompleteness Theorem does not show that T is incomplete.

It is clear that the first theorem destroys the hopes of achieving the first goal of Hilbert's Program, the axiomatizations. A little more refined argument is needed to show that the second incompleteness theorem kills also the second goal: proving the consistency of axiomatic systems used for the foundations of mathematics. To prove the consistency of such systems seems a more modest goal. We know we cannot completely axiomatize structures such as the natural numbers and the sets, but we may be satisfied with a nice and sufficiently strong axiomatic system. The fact that we are not able to completely axiomatize these structures, however, shows that we are not quite sure what are the theories that describe true arithmetic, true set theory etc. The more arbitrary the axiom systems look, the more pressing the question about consistency is. The last thing that we would want is an inconsistent system, a system in which everything is provable, which gives us no information whatsoever. In particular, in set theory we want to be sure that there will be no new generation of paradoxes that will force us to abandon the currently used axiomatic system.

Let us return to Hilbert at the beginning of the 20th century to see how the second incompleteness theorem works. Based on the original works of Frege, Russell, Whitehead, and others, Hilbert's proof theory established that a proof is a finite mathematical structure described by elementary combinatorics. So the mathematics needed for formalizing the syntax of logic is very simple. The consistency of a theory is simply another syntactical concept. Thus Hilbert was naturally led to the belief that the consistency problem can be handled with the same sort of mathematics. Why would one need to use infinite sets to prove something about finite sets? Hilbert had been thinking about the problem for several years, thus it was clear to him that proving consistencies would not be a completely trivial thing, and he knew very well that he had to use some assumptions, some means. He never specified precisely his idea of *finitary means* that he proposed for proving the consistencies, but the name tells us enough: one should only use numbers, finite structures, etc., infinite sets were disallowed. He hoped that with such restricted means one could eventually prove the consistency of arbitrary consistent theories, even theories such as Zermelo-Fraenkel set theory, in which there are extremely large infinite sets. Suppose we can present the finitary assumptions that he had in mind as a formal system,

[15] November 20 and November 17, 1930 respectively, see [58], p. 87. Gödel's priority has never been seriously challenged. The only fact worth mentioning is that in 1905, Poincaré conjectured (without stating it formally) that one cannot prove the consistency of mathematical induction without using mathematical induction itself [220]. Gödel's Second Incompleteness Theorem applied to Peano Arithmetic implies a possible formal version of Poincaré's conjecture, but it is even stronger.

a theory T based on first-order logic. Then Gödel's second theorem tells us that T is not capable of proving its own consistency. So T already fails to prove the consistency of a theory that only talks about finite objects, let alone the consistency of a theory that talks about infinite ones.

We may modify the approach by saying: well, we cannot only use finitary means, so let's look for *any* theory that would prove the consistency of all consistent theories. But Gödel's theorem also prevents this; in fact, Gödel's theorem says exactly that this is not possible. Every theory T fails to prove the consistency of some theory, namely of itself. Note that if a theory S contains theory T, then the consistency of S immediately implies the consistency of T. Thus T not only does not prove the consistency of itself, but it also does not prove the consistency of any theory that contains T.

Another consequence of the incompleteness theorem is that we can always expand our theory by adding to it the statement of its formal consistency as a new axiom. The expanded theory again does not prove its consistency, it only proves the consistency of the original one. (But, perhaps, this may be a way to produce new useful axioms? I will elaborate on this in Chap. 7.)

Gödel's theorems were the end of Hilbert's Program, but they did not destroy proof theory. This field is still flourishing and results related to the incompleteness theorems are an important part of it.

After this brief acquaintance with the incompleteness theorems, I defer the proofs to Chap. 4, where I will also explain the theorems in more detail. It is impossible to fully understand the incompleteness theorems without knowing at least the basic ideas of proofs.

Misconceptions About the Incompleteness Theorems

The incompleteness theorems are very important results and many people outside of logic know them, but they are often wrongly interpreted. The most frequently occurring misinterpretation concerning the first incompleteness theorem is that '*there are unprovable sentences*', meaning that there are mathematical theorems that can never be proved. The mistake is in that Gödel's theorem does not claim an *absolute* impossibility of learning truth; it only says that *relative* to a theory something is impossible to prove. Take an arbitrary true sentence ϕ. Then, trivially, there is a theory in which it is provable, namely the theory whose axiom is ϕ. Or, if you do not like this, add ϕ to the currently used axioms of set theory. Thus the theorem does not exclude that we can gradually expand our axiom systems and the resulting theories can potentially prove any given sentence. Nowadays most mathematicians accept Zermelo-Fraenkel set theory as the foundations of mathematics, so one may suggest interpreting '*provable*' as '*provable in Zermelo-Fraenkel set theory*'. The problem is, however, that it is more natural to accept this set theory as an open system that we can gradually expand by new axioms. A natural way to expand Zermelo-Fraenkel set theory is to add large cardinal axioms, which are a kind of higher infinity axioms (I will consider this topic in the following chapters). What the incompleteness theorem does say is that we cannot expand theories in a systematic way. Certainly, humankind will only be able to prove finitely many theorems, thus there will be some that we will never be proved. But we cannot tell which are those we will never prove.

A related misconception is that '*we cannot prove the consistency of the foundations of mathematics*'. Suppose we only needed finite structures, so we could use Peano Arithmetic or Finite Set Theory (defined on page 116). For proving the consistency of these theories, it suffices to use essentially any set theory that postulates the existence of infinite sets, much less than the full strength of Zermelo-Fraenkel Set Theory. This also explains '*the paradox that we know that formal arithmetic is consistent, but we cannot prove it*'. Our belief in the consistency of formal arithmetic (represented by Peano Arithmetic) is based on the fact that we need very simple set-theoretical assumptions to construct its model. The correct statement is that *we cannot prove the consistency of some formal foundations of mathematics in the same, or a weaker system.* This is, in fact, the Second Incompleteness Theorem restated in different words.

Here I should remind the reader that finding the foundations of mathematics in which their consistency would be provable was *not* the goal of Hilbert's Program. Having a formal system T in which its consistency would be provable would not help us to justify its consistency. If one believes that theorems of T are true, then one implicitly assumes that T is consistent. Therefore, if we use some T as the foundations, it does not matter whether the consistency of T is provable in T.

The main problem in understanding the incompleteness theorem stems from the fact that the theorem, along with asserting that there is an unprovable sentence, also gives us a concrete true sentence that is missing in the theory (the consistency of the theory). Therefore, it seems that while theories are incomplete, this phenomenon does not affect our knowledge of the true facts. One is lead to the conclusion that we, people, have the ability to overcome the limitations posed on formal systems by the incompleteness theorem. A careful analysis of such arguments immediately reveals the fallacy. Essentially, one has only to spell out the assumptions that are used. These arguments are always based on the assumption that the theory in question is sound, which means that it only proves true sentences. So the fact that we can consistently extend the theory by adding the Gödel sentence to it is already contained in the assumption. This, certainly, needs a longer discussion, which I defer to Chap. 7.

Shortly after the incompleteness theorems were published, there were attempts to avoid the incompleteness phenomenon by replacing first order theories by something else, but very soon it turned out that this cannot work. The reason is that the incompleteness of the theories to which Gödel's theorem is applicable is caused by the complexity of the sets of sentences provable in these theories. Every theory in which it is possible to prove certain elementary propositions about numbers must have this complexity and, therefore, cannot be complete. Consequently, it is not possible to avoid the incompleteness by changing logic and formalism.

On the Proof of the Completeness Theorem

Proving completeness is not as trivial as proving soundness, but once the concepts involved are sufficiently familiar, the proof is not difficult. To prove the theorem Gödel considered the contrapositive implication: if a sentence does not have a proof, then it is not true in all structures. Thus we only need, for a given unprovable sentence, to construct a structure that does not satisfy the sentence: it satisfies its negation. Consider such a hypothetical structure M that satisfies the negation of the sentence. What else should the structure satisfy? Certainly, all sentences that logically follow from the negation of the sentence. This still leaves a lot of sentences that we do not know whether they should or should not hold in M because the sentence does not decide everything. Here comes the first trick. If we cannot decide a sentence φ we can (expressing it in trendy terms) break the symmetry by choosing φ or *not* φ arbitrarily. In both cases the enlarged set of sentences will be consistent. The enlarged set will imply more sentences, but there will still be some undecided. So we repeat this enlarging process on and on. Doing it in a systematic way will ensure that after infinitely many steps we decide all sentences and still have a consistent theory of what should hold in M.

Now we have all sentences that should be true in the hypothetical model M, but we still do not have the model. The crucial idea of this proof is that we can use logical symbols as the elements of the structure. I have warned about the distinction between syntax and semantics so you may be worried that this would break this law, but what Gödel did is permissible. The logical calculus is a mathematical entity and lives in the same world as structures. We are now looking at this world from the outside and consider relations between the elements of the logical calculus and structures. So both proofs and models are the subject of our study, they are at the same level, they are in the world of structures. Proofs, as sequences of symbols, are simply another type of structures. We know that the substance of the elements of structures is irrelevant, so the elements of the structure M may coincidentally be symbols used in the calculus. To construct M, we assume that we have sufficiently many constant symbols and take the universe of M to be these symbols. More precisely, we have to ensure that for every existential statement, we have a constant which testifies this statement. Then it remains to define relations and operations. This turns out to be easy, as we already have all the sentences that should be true in M. To decide, for instance, whether c should be in relation R with d, we just look at the theory and see whether the corresponding sentence $R(c, d)$ is there.

See Notes for more detail on the proof of the Completeness Theorem.

Constructive Mathematics—Proofs Instead of Structures

As it is natural to assume that there is the real world that we perceive by our senses and about which we reason, it is also natural to think about mathematics in this way. Therefore, I have started with describing structures and went on by describing logic as a means of studying structures. But mathematical reality is elusive. When we study the natural numbers, we do not perform experiments, instead we use axioms, for example, the induction axiom. We can try a few examples to see if a conjecture is reasonable, but the only way we can convince ourselves that it is a theorem is to prove it. A proof may use facts that we know about other structures, or facts about sets, but if we analyze carefully all the assumptions that we use, we find out that we are proving it from basic axioms such as axioms for sets and induction. Thus, while pretending that we are working with structures, we in fact are just doing proofs. Hence, shouldn't we consider a proof as the basic entity in mathematics, all the rest being derived from it?

There is a stream in mathematical logic that pursues such a direction; it is called *constructive mathematics*. The main sources from which it originated are the *intuitionism* represented by the Dutch mathematician L.E.J. Brouwer (1881–1966) and the *constructivism* of the Russian mathematician Andrey A. Markov Jr. (1903–1979). Intuitionism started around 1900 as a reaction to two events. The first was the emergence of nonconstructive proofs, which are proofs that show the existence of a mathematical object without explicitly constructing it. The second was the so called foundation crisis caused by the discovery of Russell's paradox. Intuitionists claimed that paradoxes were just samples of symptoms of the ill foundations of mathematics.

The most prominent figure among those whose views were connected with intuitionism at its early stages was the French mathematician Henry Poincaré (1854–1912). Very soon Brouwer became the most important proponent of intuitionism and it was he who coined the name of this stream.

Intuitionists proposed to found mathematics not on formal systems, but on mathematicians' intuition. Their argument does have some appeal: whatever problems occurred in the foundations, they had little impact on everyday mathematics. Most concepts, originally based solely on intuition, survived all the foundation crises without problems. While formal foundations always have had some trouble, intuition apparently hasn't. This argument is acceptable, but if one rejects the classical set theoretical foundations of higher order structures, such as the real numbers, one has to propose something else. Therefore, Brouwer started an ambitious program of revising the whole of mathematics in an intuitionistic manner. What he proposed was too restrictive for most mathematicians and they did not accept it.

One of the main differences between classical mathematics and constructive mathematics is logic. Describing logic went against the aims of intuitionism, as they reject formal systems in favor of intuitive reasoning, but it turned out that the logic they used is a very natural system. It is now called *intuitionistic logic*. Intuitionistic logic uses the same connectives and quantifiers but some of them are interpreted differently. In particular, the disjunction $\phi \vee \psi$ is interpreted as follows. One can assert $\phi \vee \psi$ to be true, only if one can assert that ϕ is true or that ψ is true. In classical logic we can assert $\phi \vee \psi$ without knowing which of the two propositions is true; in intuitionistic logic it is not allowed. The main example is the *law of excluded middle*, which is the classical tautology $\phi \vee \neg\phi$. This is true, in classical logic, no matter whether or not we know which of the propositions ϕ and $\neg\phi$ is true. In intuitionistic logic we must know which of the two is true. Thus one of the basic classical laws is rejected in intuitionistic logic.

Example In intuitionistic mathematics a real number is identified with a sequence of rational numbers that approximate it with arbitrary precision. If we are given two real numbers r_1 and r_2 by such sequences, we do not know how far we must go to establish that $r_1 < r_2$ or $r_1 = r_2$ or $r_1 > r_2$. (In fact, to establish that $r_1 = r_2$ no approximations suffice; we need a proof.) Therefore the trichotomy law does not hold for real numbers in intuitionistic mathematics.

Here are examples of some pairs that start as if they were the same.

$$\begin{aligned} \pi &= 3.14159265\ldots \\ 355/113 &= 3.14159292\ldots \end{aligned}$$

$$\begin{aligned} \sqrt{5}+\sqrt{6}+\sqrt{18} &= 8.928198\ldots \\ \sqrt{4}+\sqrt{48} &= 8.928203\ldots \end{aligned}$$

These examples concern two important problems studied in mathematics and computer science: 1. the problem of rational approximations of real numbers, 2. the problem about the complexity of deciding inequalities between sums of square roots of integers.

Intuitionists distinguish two relations that are the same in classical mathematics: *non-equality* $r_1 \neq r_2$ and *apartness* $r_1 \# r_2$. The first means that we only know that r_1 cannot be equal to r_2, the second means that we have an estimate on how much the two real numbers differ. Thus knowing that π is irrational, we can conclude that $\pi \neq 355/113$; to establish that $\pi \# 355/113$ we need a concrete estimate such as $355/113 - \pi > 0.0000002$.

At first glance intuitionistic interpretation of disjunction looks as if it completely eliminates this connective. Why should we assert $\phi \vee \psi$ when we know that ϕ is true? But it is not so. When combining disjunction with other connectives we can get intuitionistic tautologies that use disjunction essentially. Consider the proposition $\alpha \vee \beta \rightarrow \phi \vee \psi$. The intuitionistic interpretation of this proposition is: if one can decide which of α and β is true, then one can decide which of ϕ and ψ is true. Hence, for instance, $\phi \vee \psi \rightarrow \phi \vee \psi$ is an intuitionistic tautology.

Similarly the existential quantifier is reinterpreted in intuitionistic logic. In order to be able to assert that there exists an element x with a property ϕ, one has to provide an example of such an element. This is quite a natural requirement when one interprets disjunction in the way above, since the existential quantifier is like an infinite disjunction. Disallowing the law of excluded middle is a drastic restriction, as it results in disallowing proofs by cases and proofs by contradiction. But intuitionistic interpretation of the existential quantifier is not so alien to a working mathematician. Mathematicians have always preferred constructive proofs of existential statements. ‘Constructive’ means that an element with a given property is somehow constructed explicitly. Nonconstructive proofs, proofs in which the existence of an element is deduced indirectly, appeared only at the end of the 19th century. At that time some mathematicians rejected such proofs. I will mention some specific nonconstructive results in Chap. 5 (see page 391).

Intuitionistic logic prohibits certain proofs, so one may hope that foundations based on this logic would be safer. However, Russell's Paradox can be derived in this logic too, therefore changing only logic is not sufficient. While intuitionistic logic is a very natural and stable concept, the other changes proposed by intuitionism are not so well justified. In fact research is still going on with the aim of producing constructive foundations that could compete with classical ones. Nevertheless, there are some interesting and very abstract structures that are based on intuitionism.

The Russian constructivist school came later, in the late 1940s. It was based on more formal concepts. It shares with intuitionism the intention to prohibit nonconstructive reasoning and to this end it uses the same logic. But unlike intuitionism, it gives a precise meaning to the disjunction and the existential quantifier, a meaning based on the concept of algorithm. For example, a sentence

For every x, $P(x)$ or not $P(x)$.

is true if and only if there is an algorithm to decide the property P. Or one can say that

For every x, there exists y such that $Q(x, y)$.

only if it is possible to construct y from x such that $Q(x, y)$, which precisely means that there is an algorithm that does it. The mathematics of constructivism is based on the strict requirement that only constructive structures should be studied. Again, 'constructive' means algorithmically computable. As a result all real valued functions must be computable in the following sense. Given a good approximation of x by a rational number r, we should be able to compute a good approximation of the function value $f(x)$ by a rational number q. A consequence is that all such functions are continuous. This is an appealing feature, as it returns to the original intuition of the function and very well corresponds to what is going on in reality. In the real world all physical processes are more or less continuous. But such little positive pieces are completely overwhelmed by a host of negative results such as the result saying that there is a piecewise linear function that is not integrable.

Thus most mathematicians do not find the constructivists' approach to be very useful. Although constructive mathematics has contributed a lot to the study of computations, it seems that more has been done in the classical fields of mathematics; this concerns especially computational complexity. On the positive side, constructive systems are very useful for designing programming languages and in automated theorem proving.

Intuitionistic logic can also be used to obtain new theorems in classical mathematics. Remarkable examples of such applications can be found in the area of *proof mining*. This is a research program whose goal is to extract more information from existing proofs. It was initiated in the 1950s by Georg Kreisel who asked the famous question: "*What more do we know if we have proved a theorem by restricted means, than if we merely know that it is true?*" Many logicians have contributed to this program (for example, J.-Y. Girard analyzed Van der Waerden's Theorem [93], H. Luckhardt analyzed Roth's Theorem [187]). U. Kohlenbach studied the logical structure of certain proofs in functional analysis and succeeded in strengthening and generalizing a number of theorems in this field [158]. The key tool that he used was intuitionistic theories. In spite of using intuitionistic systems, the results he obtained are standard mathematical theorems that have nothing to do with intuitionism.

There are various branches of constructive mathematics, but most logicians working in constructive mathematics share the view that the principal objects in mathematics are proofs. For Brouwer, mathematics was a mental activity independent of any objective reality and the products of this activity were proofs. Some constructivists go as far as to reject semantics at all. Contemporary constructive mathematics focuses on connections between algorithms and proofs. The connections discovered so far confirm the key role of proofs in mathematics. This is in contrast with the traditional view that the main objects are mathematical structures and proofs only serve to gain more knowledge about them.

Notes

1. *Algebra of logic and the logic of relations.* George Boole (1815–1864) partially realized Leibnitz's idea in his book *An Investigation of the Laws of Thought.* He axiomatized the theory of classes with the basic set-theoretical operations, intersection, union and complement. From the point of view of the systems currently used in mathematical logic, we can view his system either as propositional logic, or first-order logic restricted to the use of predicates, i.e., unary relations, or as an axiomatization of Boolean algebras. Logic, however, cannot be based only on unary relations. Therefore, Ernst Schröder (1841–1902), Charles S. Pierce (1839–1914) and others developed *algebras of relations*. This line of research culminated in the work of Tarski and his students, who introduced the concept of *cylindric algebras* [119, 120]. Cylindric algebras formalize first-order logic in a purely algebraic fashion, like Boolean algebras do in the case of propositional logic.
2. *Axioms and rules for propositional logic.* The first thing we should realize is that there is a simple algorithm to test whether a propositional formula is a tautology. By definition, a formula is a tautology if and only if it is true for all truth assignments to propositional variables. Since every variable can only have two values there is only a finite number of truth assignments and one can systematically check all of them. Why then do we need any axiomatization at all? There are several reasons. One reason is purely aesthetic: we have to axiomatize first-order logic, so it would be awkward if the propositional part of it was presented in a different way. Another one is that the way people think is closer to an axiomatic system than to truth value checking. Last but not least, for propositions with many variables, the exhaustive algorithm would run too long, while the proposition may still have a short proof. One of the main open problems in proof complexity is whether or not there exists a proof system for propositional logic in which every tautology has a proof of at most polynomial length.

   Here are the axiom schemata of a system of axioms for propositional logic (from [110]).

$P \to (Q \to P)$
$(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$
$(P \vee Q) \to (Q \vee P)$
$(P \wedge Q) \to (Q \wedge P)$
$P \to (P \vee Q)$
$(P \wedge Q) \to P$
$(P \to (Q \to (P \wedge Q))$
$((P \to R) \wedge (Q \to R)) \to ((P \vee Q) \to R)$
$(P \to (Q \wedge \neg Q)) \to \neg P$
$(P \wedge \neg P) \to Q$
$P \vee \neg P$

The system has a single rule, modus ponens:

From P and $P \to Q$ derive Q.

In the above axiom schemata and in the rule the letters P, Q, R are variables representing arbitrary propositions. In the propositional calculus we can substitute for them arbitrary propositional formulas to get an instance of an axiom or the rule. Thus the formulas above represent infinitely many axioms.

Examples 1. Let us interpret P as the propositional variable p and Q as $p \to p$. Then $p \to ((p \to p) \to p)$ is an axiom by the first axiom schema.

2. Interpret P as p, Q as $p \to p$ and R as p. Then

$$\big(p \to \big((p \to p) \to p\big)\big) \to \big(\big(p \to (p \to p)\big) \to (p \to p)\big)$$

is an axiom by the second axiom schema. From this, we can derive $p \to p$ by applying two instances of the first schema and twice modus ponens.

A proof in this system is a sequence of propositional formulas in which every formula is either an instance of an axiom schema or follows from two formulas occurring before by the rule. Any formula in a proof is a tautology.

There are many ways in which the propositional calculus can be axiomatized. The above one belongs to a group in which the aim is to minimize the number of rules, namely, there is only one rule and many axiom schemata. In such systems modus ponens is the most common rule used. The other extreme is to take a minimal number of axiom schemata with many rules. Such systems are called *sequent calculi* and are extremely useful for analyzing proofs. Note that due to the fact that in the axiom schemata above the main connective is the implication, one can easily present them as rules (except for the last one).

Axiomatizations also depend on the set of connectives that we want to use. We can take a small set of connectives that forms a complete set of connectives (for example, $\neg$ and $\to$) and thus we get a simpler set of axioms.

There are several other types of systems that are not based on axioms and rules in particular natural deduction systems, see below.

3. *Axioms and rules for first-order logic.* To get an axiomatization for first-order logic, we can take the set of axiom schemata from the previous paragraph, *modus ponens* (now applied to first-order formulas) and add only two axioms and two schemata specific for quantifiers. The axiom schemata are:

   $(\forall x \phi(x)) \to \phi(t)$
   $\phi(t) \to \exists x \phi(x)$

   where x stands for an arbitrary first-order variable and t stands for an arbitrary term whose variables are not quantified in ϕ and it is substituted for all occurrences of the variable x. The rules are:

   From $\phi \to \psi(x)$ *derive* $\phi \to \forall x \psi(x)$.
   From $\psi(x) \to \phi$ *derive* $(\exists x \psi(x)) \to \phi$.

   In both rules there is a restriction that the variable x must not occur in ϕ. (One can weaken it to: every occurrence of x in ϕ is in the scope of a quantifier that binds x).

   Formal systems of this form are called *Hilbert style systems.*

4. *Natural deduction systems.* The main feature in which they differ from others is that a proof may contain a subproof. A subproof uses some additional hypotheses that are valid only in the subproof. An example is a subproof where an implication $\alpha \to \beta$ is proved. The subproof starts with the hypothesis α and ends with β. The subproofs may contain other subproofs and so on. One can use formulas from outer subproofs in inner subproofs, but not conversely. This is very much like in programming languages where one can use procedures with local variables. Most rules come in pairs associated with connectives or quantifiers. One member of the pair is used to obtain a more complex formula using the connective or the quantifier, the other does the opposite.

   Here is a list of the axioms and rules of a natural deduction proof system.
   Axiom schema: $A \vee \neg A$.
   Rules:

   a. *From* A *and* B *derive* $A \wedge B$.
   b. *From* $A \wedge B$ *derive* A *and* B.
   c. *From* A *derive* $A \vee B$ *and* $B \vee A$.
   d. *If* C *was derived from hypothesis* A *in a subproof and* C *was also derived from hypothesis* B *in a subproof, then from* $A \vee B$ *derive* C.
   e. *If* B *was derived from hypothesis* A *in a subproof, then derive* $A \to B$.
   f. *From* A *and* $A \to B$ *derive* B (*modus ponens*).
   g. *If* B *and* $\neg B$ *were derived from hypothesis* A *in a subproof, then derive* $\neg A$.
   h. *If* B *and* $\neg B$ *were derived from hypothesis* $\neg A$ *in a subproof, then derive* A (*proof by contradiction*).
   i. *From* $A(y)$ *derive* $\forall x A(x)$ (y *must not occur in any hypothesis*).
   j. *From* $\forall x A(x)$ *derive* $A(t)$ (*for any term* t *whose variables are not quantified in* A).
   k. *From* $A(t)$ *derive* $\exists x A(x)$ (*for any term* t *whose variables are not quantified in* A).

1. *If B was derived from hypothesis $A(y)$ in a subproof, then from $\exists x A(x)$ derive B (y must not occur in other hypotheses).*

The first natural deduction system was introduced by a Polish logician S. Jaśkowski in 1934.

5. *The proof of the Completeness Theorem—more detail.* Recall that given an unprovable sentence ϕ we want to construct a model M in which ϕ is false. Note that ϕ being unprovable is equivalent to $\neg\phi$ being consistent. So, for a consistent sentence we need to construct a model. I will consider a more general task: for a given consistent set T of sentences to construct a model in which they are satisfied. Thus, in particular, I will show that every consistent formal theory has a model.

The idea is to gradually enlarge the set T to a set T' in such a way that we can use terms of the language as the elements of a model. For the sake of simplicity, let us assume that the language of T contains one binary relation R and, possibly but not necessarily, function symbols and constants; furthermore, we will assume that equality is not in the logical calculus.

I will start by stating the properties that T' should satisfy:

a. T' contains T;
b. T' is consistent;
c. T' is complete;
d. for every sentence $\exists x\varphi(x)$ in T', there exists a closed term (a term without variables) t such that $\varphi(t)$ is in T'.

Given T' with the above properties, define a model M as follows. The universe consists of all closed terms of T'. The interpretation of R in M, denoted by R^M, is defined by

$$R^M(s,t) \quad \text{if and only if} \quad R(s,t) \text{ is in } T',$$

where s and t are terms without free variables. Similarly, the interpretation of an n-ary function symbol F, denoted by F^M, is defined by

$$F^M(t_1,\ldots,t_n)=s \quad \text{if and only if} \quad F(t_1,\ldots,t_n)=s \text{ is in } T'.$$

The main lemma is the following:

Lemma 1 *A sentence is true in M if and only if it is in T'.*

This lemma is easily proved by induction on the logical complexity of the sentence (the number of connectives and quantifiers). The base case, which is the case of atomic formulas, follows immediately from the definition of M.

Consider $\neg\varphi$, and suppose that the lemma is true for all sentences of smaller complexity. In particular, the lemma must hold for φ. By the consistency and the completeness of T', $\neg\varphi$ is in T' if and only if φ is not in T'. Thus we get the lemma for $\neg\varphi$.

Now suppose we want to prove it for $\exists x\varphi(x)$ assuming it holds for all sentences of smaller complexity. If $\exists x\varphi(x)$ is in T', then for some term t, $\varphi(t)$ is in T'. Since we assume the lemma for formulas of smaller complexity, $\varphi(t)$ is true in M, hence also $\exists x\varphi(x)$. Proving the converse implication, assume that $\exists x\varphi(x)$ is true in M. Then, for some closed term t, $\varphi(t)$ is true in M. By the induction assumption, this implies that $\varphi(t)$ is in T'. Hence (by the completeness and the consistency of T') also $\exists x\varphi(x)$ is in T'. I leave the other cases to the reader.

To finish the proof of the completeness theorem, it is sufficient to construct T'. Indeed, if we have such a T', then the model M from the lemma is a model of T by the first condition. The construction of T' is done by an infinite process in which we at each step either add a sentence φ if it is still independent, or we add a new constant symbol c and a sentence $\varphi(c)$ if $\exists x\varphi(x)$ is already in the set so far constructed. The technical details how to do it are inessential.

6. *Compactness.* The most important property of first-order logic, called *compactness*, is a consequence of having finitary syntactical means. It is the following fact:

Compactness of First-Order Logic *A set of sentences Φ is consistent if and only if every* finite *subset of Φ is consistent.*

The proof of this proposition is easy. Φ is inconsistent means that there exists a proof of contradiction using Φ as assumptions. But a proof is a *finite* sequence of formulas, hence it only uses a finite number of sentences from Φ. Note that there are also (more difficult) proofs that do not use the Completeness Theorem.

As an easy application, I will show that an infinite model M can always be expanded to a larger model M' (which means that the universe of M' is a proper superset of the universe of M) so that M' satisfies the same sentences as M. To this end, let Φ be the set of all sentences true in M. Let us introduce a symbol, a constant, for every element of M; let c_a, $a \in M$ be these constants. Let c be another constant symbol. Now we are using a language with infinitely many symbols, but all the facts that we need hold true also in this case. Let Ψ be Φ augmented with axioms $c \neq c_a$, for all $a \in M$. I claim that Ψ is consistent. Suppose not, then we have a finite set Ψ' of sentences from Ψ that are inconsistent. Those finitely many sentences can only use a finite number of constants c_a. But then M is also a model of Ψ' because we can interpret the new constant c as an element not mentioned in Ψ'. Hence Ψ' is consistent. By the Completeness Theorem it has a model M'. The universe of the model M' does not have to contain the universe of M. However, as we have a constant for each element of M and these constants are interpreted in M' we have a natural embedding of M into M': element a of M is mapped on the interpretation of c_a in M'. Since M' satisfies all new axioms $c \neq c_a$, the interpretation of c is different from the interpretations of constants c_a. Hence no element of M is mapped on the interpretation of c.

The constructed model M' satisfies the following stronger property: if ϕ is a formula with k free variables, and $a_1, a_2, \ldots, a_k$ are elements of M, then $M \models \phi[a_1, a_2, \ldots, a_k]$ if and only if $M \models \phi[a_1, a_2, \ldots, a_k]$. Such a model M' is called an *elementary extension* of M. If we start with the standard model of arithmetic $(\mathbb{N}; +, \cdot, \leq)$, we obtain a nonstandard model in which exactly the same arithmetical sentences are true as in the standard one.

One can perform this construction with more than one c, in fact, with an arbitrarily large set of constants. If one adds also the axioms $c \neq c'$ for every pair of new distinct constants, then a model of arbitrary large cardinality is obtained. This is the *upward* version of the Löwenheim–Skolem Theorem.

7. *Peano Arithmetic and Finite Set Theory.* One of the most studied theories in mathematical logic is *Peano Arithmetic* (abbreviated as *PA*). It is a theory whose language has one constant 0, one unary operation S and two binary operations $+, \cdot$.[16] The operation $S(x)$ is interpreted as $x+1$, the successor of x. It is axiomatized by the following axioms:[17]

$$\begin{aligned}
&S(x) \neq 0\\
&S(x) = S(y) \rightarrow x = y\\
&x \neq 0 \rightarrow \exists y(x = S(y))\\
&x + 0 = x\\
&x + S(y) = S(x+y)\\
&x \cdot 0 = 0\\
&x \cdot S(y) = x \cdot y + x
\end{aligned}$$

and for every formula $\phi(x)$, the following is an axiom

$$\big(\phi(0) \wedge \forall x\big(\phi(x) \rightarrow \phi\big(S(x)\big)\big)\big) \rightarrow \forall x\, \phi(x).$$

This is called the *induction axiom schema* for arithmetical formulas.

The first seven axioms of Peano Arithmetic is a theory called *Robinson Arithmetic*.

Robinson Arithmetic is usually presented with the binary relation $\leq$ and an axiom, which is, in fact, a definition of this relation:

$$x \leq y \equiv \exists z(z + x = y).$$

This theory is very weak (it does not prove even such basic facts as the commutativity and associativity of the operations and the transitivity of $\leq$), but is important in logic.

In algebra one uses the constant 1 instead of the successor function from which the successor function is definable. The successor function is only used in logic and some programming languages. The reason for using it explicitly in axiomatizing Peano Arithmetic is that it is the most primitive function in this system. Note that the axioms for $+$ and $\cdot$ are, in fact, the natural recursive definitions of these operations. Thus it is more natural to have the successor function explicitly. Note, however, that the axioms for addition and multiplication are not definitions in the sense of first-order logic. We can use these formulas as definitions only in higher order logics, or in set theory.

Peano Arithmetic is not quite suitable as a theory on which we could base the foundations of mathematics because the only objects in it are numbers. Even elementary mathematics needs set theory; one should be able to talk at least

[16] In the sequel I will also often denote multiplication by $\times$.

[17] I am using the standard convention that the universal quantifiers in front of formulas are omitted.

about finite sets. Although we cannot talk directly about finite sets in Peano Arithmetic, it is possible to code sets by numbers. One of the possible encodings is the following. For a number n, define a set D_n by $D_0 = \emptyset$ and for $n > 0$ let

$$D_n = \{D_{k_1}, \ldots, D_{k_m}\}$$

where $n = 2^{k_1} + \cdots + 2^{k_m}$, $k_1 < \cdots < k_m$. This mapping is a bijection between the natural numbers and *the hereditarily finite sets*. These are sets that are finite, its elements are finite, the elements of its elements are finite, etc. Having this coding we can speak freely about finite structures in Peano Arithmetic. Consequently, we can talk about formulas and proofs, about Turing machines, etc.

Let us now consider the structure consisting of hereditarily finite sets with the usual membership relation $\in$. In the same way, as the natural numbers are the canonical structure for number theory, this is a canonical structure for theories of finite sets. There are several equivalent axiomatic systems whose model is this structure. One such system is the usual Zermelo-Fraenkel set theory with the axiom of infinity replaced by its negation, an axiom saying that all sets are finite. But in fact we do not have to take all the axioms of Zermelo-Fraenkel; for instance, the Axiom of Choice can be derived from others in this theory. The theory is denoted by ZF_{fin} and I will call it *Finite Set Theory* (for lack of a better name). The problem with this name is that it suggests that the theory completely describes the structure of hereditarily finite sets, which is impossible by the incompleteness theorem. But this theory is such a natural system that the name is justified. In particular, if we translate its axioms using the above encoding, we get arithmetical statements provable in Peano Arithmetic. Hence, everything that we can prove in Finite Set Theory can also be proved in Peano Arithmetic. And vice versa, if we define the natural numbers in Finite Set Theory in the usual way, then we can prove all the theorems of Peano Arithmetic. Thus Peano Arithmetic and Finite Set Theory are the same theories up to suitable translations.

This shows that Peano Arithmetic and Finite Set Theory are very natural theories.

8. *Proving relative consistency by interpretation.* We say that a theory T is interpretable in theory S if it is possible to represent the relations and function symbols of T by the formulas of S so that the translations of the axioms of T are provable in S.

I will define it more precisely in the special case in which the language of T contains only one binary relation R. Let ρ be a formula in the language of S with two free variables. Then for every formula ϕ in the language of T, we can translate ϕ into a formula of the language of S by replacing all atomic subformulas of the form $R(x, y)$ by formulas $\rho(x, y)$. We say that ρ defines an interpretation of T in S, if the translations of the axioms of T are theorems of S.

Notice that the last condition implies that the translations of all theorems of T are theorems of S. In particular, if T is inconsistent, then so is S. Thus we get an easy, but very important fact:

Theorem 3 *If S is consistent and T is interpretable in S, then T is also consistent.*

Proving that T is interpretable in S is a very efficient way of proving the relative consistency of T with respect to S, especially when T is axiomatized by a finite set of axioms. Then the proof of the relative consistency is given by the proofs of the translations of the axioms. Thus we have a completely finite object that shows the relative consistency.

Usually we need a slightly more general concept of interpretation, in which the domain of the interpretation of T may be a proper subdomain of S. The above relation between Finite Set Theory and Peano Arithmetic are such mutual interpretations. Many relative consistency proofs in set theory are also by interpretation.

9. *A consistent theory that proves its own inconsistency.* According to the Second Incompleteness Theorem a consistent theory T (and sufficiently strong to express such things) does not prove its consistency. This is equivalent to the following statement: the theory T augmented by the statement that T is inconsistent is consistent. I will use $+$ to denote the operation of adding an axiom to a theory and denote by Con_T the formal statement that T is consistent. Thus the Second Incompleteness Theorem can be equivalently stated as follows:

   If T is consistent, then $T + \neg Con_T$ is consistent.

   Now, let T be consistent. Then also $T + \neg Con_T$ is consistent and this theory proves that T is inconsistent. Since $T + \neg Con_T$ is an extension of T, if T is inconsistent, $T + \neg Con_T$ is inconsistent too. This can be formalized in $T + \neg Con_T$, hence this consistent theory proves its own inconsistency.

   What are models of this theory? By the Completeness Theorem we know that this theory has a model. The natural numbers of the model cannot be the standard model, since in the standard model there is no inconsistency of T. Thus the numbers of this model form a nonstandard model and the number representing the proof of contradiction in T is a nonstandard number.

10. *Reducing the consistency of Peano Arithmetic to the well-ordering of ε_0.* In 1936, not long after Gödel published his seminal results on incompleteness, results that essentially destroyed Hilbert's Program, Gentzen obtained a fundamental result in proof theory. His result is in the direction that Hilbert was aiming at: it gives a proof of the consistency of Peano Arithmetic using an argument that seems acceptable from the point of view of finitism [91, 92]. Gentzen proved the consistency of Peano Arithmetic using transfinite induction over ε_0. The ordinal ε_0 is a more complex structure than the natural numbers, but it can be presented as efficiently as the natural numbers, including the operations of addition, multiplication and exponentiation (see Chap. 3).

    It is disputable whether Gentzen's result is a realization of Hilbert's Program in the special case of Peano Arithmetic, but in any case it is an important result. As far as the consistency problem is concerned, it gives us some justification to believe that PA is consistent, as the consistency is based on an assumption that is not directly linked with PA. Another consequence is that we get an independent sentence that is different from the one given by the second incompleteness theorem and which is more "mathematical".

    I will say more about this subject in Chap. 6.

11. *Automated proof checking and theorem proving.* I will explain later why it is not possible to automate the problem of finding a proof of a theorem. Since mathematicians are able to prove difficult theorems nevertheless, and everybody (including animals) can do more or less difficult logical reasoning, it should be possible to do something also on computers. People had wondered to what extent our thinking can be simulated by computers even before the first computers were constructed. Nobody knows how far we are from the day that one will be able to say that a computer thinks, but it is, perhaps, the biggest challenge of humankind.

    Automated theorem proving started in the 1950s with systems designed by M. Davis, P.C. Gilmore, H. Wang and others. It is interesting to note that some of them were quite successful though they had been written before key concepts were introduced in the area (namely the *Davis-Putnam procedure* of Martin Davis and Hilary Putnam, in 1960 [56], and the *resolution*[18] of John Alan Robinson, in 1965 [248]). One of the successes of automated theorem proving is a proof of *Robbins' Conjecture*. This conjecture, which is now a theorem, states that the following equations suffice to axiomatize Boolean algebras:

$$\begin{aligned} x \vee y &= y \vee x \\ (x \vee y) \vee z &= x \vee (y \vee z) \\ \big((x \vee y)' \vee \big(x \vee y'\big)'\big)' &= x. \end{aligned}$$

    This means that the equational theory defined by these equations, augmented by definitions $0 := (x \vee x')'$, $1 := x \vee x'$ and $x \wedge y := (x' \vee y')'$, is equivalent to the standard axiomatizations, such as the one given on page 21. The conjecture was stated in 1933 and was proved by W. McCune in 1996 [197] using his computer program *EQP* (related to his more well-known *OTTER*). It is not surprising that computers proved to be superior to people in this particular area. After all, manipulating with equations, which in most cases have no meaning, is a rather mechanical work. Yet it is a remarkable fact, since some very good mathematicians tried to solve the problem.

    The proof of Robbins' conjecture should not be confused with *computer assisted proofs*. In computer assisted proofs, such as the proof of the famous Four Color Conjecture, the problem is reduced to a search of a large number of special structures. Except for this part, the proof is completely designed by a mathematician. In the case of Robbins' Conjecture, a general program for proving equations was applied to the conjecture. So computer assisted proofs require formalization of only some special concepts, while theorem provers need formalization of all possible proofs in the field considered. The advantage of Robbins' Conjecture was that it only sufficed to formalize equational proofs.

    One of the first languages for formalizing proofs so that they can be used in computers was N.G. de Bruijn's *AUTOMATH*,[19] whose development started in

[18] See page 60.

[19] http://www.cs.ru.nl/~freek/aut/

1967. In 1994 an initiative *QED* (*Quod Erat Demonstandum*, an abbreviation often used to mark the end of a proof) was launched to unify forces on building a proof checker that would be accepted as the international standard. In the *QED Manifesto* many arguments for such an enterprise had been collected. Unfortunately the project died out eventually without producing significant outputs, leaving various groups to compete, as before. (You may remember the similar, but much more successful initiative called *ALGOL* in the early history of programming languages.) A remarkable event was the verification of the proof of the Four Color Theorem by B. Werner and G. Gonthier in 2004 [106]. It was done using the proof assistant *Coq*, initially developed by T. Coquand and G. Huet in 1991. The Prime Number Theorem was verified by J. Avigad and his students using the system *Isabelle* in 2004. One of the most successful on-going projects of formalizing and verifying proofs is *MIZAR*.[20] Its library of formalized and verified theorems has more than 49 000 items at the time of the writing of these lines.

12. *Intuitionistic logic.* The axioms and rules of classical logic are often chosen so that one only needs to remove something to get intuitionistic logic. This is also the case with the system on page 111; to get an axiomatization of intuitionistic logic we only need to remove the axiom of excluded middle $P \vee \neg P$. The axioms and rules for quantifiers are the same. Thus intuitionistic logic is weaker than classical logic. On the other hand one can interpret classical logic in intuitionistic logic. For classical propositional logic, it is very simple: write two negations before the sentence. So a propositional sentence ϕ is a classical tautology, if and only if $\neg\neg\phi$ is an intuitionistic tautology.

    Intuitionistic logic is more complex than classical logic. Unlike in classical logic, it is not possible in intuitionistic logic to express one of the basic connectives $\neg, \vee, \wedge, \rightarrow$ using the others. The most attractive feature of intuitionistic logic is that one can extract algorithms from certain proofs.

    The presence of double negations that cannot be reduced to a single one shows a relation to modal logics. Both ϕ and $\neg\neg\phi$ express that ϕ is true, but in $\neg\neg\phi$ with lesser emphasis. Fortunately, three negations reduce to one. The connection to modal logics is also seen in semantics.

13. *Modal logics.* Modalities are not used in mathematics; in mathematics we only use precise statements. Still it is interesting to study modal logics. Applications of these logics are in the study of human reasoning and in artificial intelligence, but also in some branches of mathematics. In mathematics we use modal logics as mathematical entities, say, formal systems, or structures, but we reason about them using classical logic without modalities. Attempts have been made to base set theory on a modal logic, in order to avoid Russell's paradox while keeping an unrestricted schema of comprehension, but such proposals will never be accepted by mathematicians (as I explained in Sect. 2.1).

    The basic modalities are *necessarily*, denoted by $\Box$, and *maybe*, denoted by $\Diamond$. Hence, for a proposition ϕ, $\Box\phi$ is read '*ϕ is necessarily true*' or simply

[20] http://mizar.org/project/

'*necessarily* ϕ' and $\Diamond\phi$ is pronounced '*maybe* ϕ'. Using negation we can define one from the other: $\Box\phi$ is equivalent to $\neg\Diamond\neg\phi$ and $\Diamond\phi$ is equivalent to $\neg\Box\neg\phi$. Unlike in classical or intuitionistic logic, there is no canonical unique modal logic. In the spectrum of systems that have been studied there are some that seem to be more natural than others. I will describe one of these here, the system called S_4, and mention another one, *Provability Logic* in Chap. 4 (page 297). I will only talk about propositional logic.

System S_4 is an extension of classical propositional logic. Hence we accept all the axioms and rules of classical logic for formulas in the language expanded by $\Box$. The other modality $\Diamond$, will be treated as an abbreviation of $\neg\Box\neg$. For example, we accept the law of excluded middle for modal formulas. If we apply this law for the formula $\Box\phi$, we get the following tautology $\Box\phi \vee \neg\Box\phi$. Using $\Diamond$, this is equivalent to $\Box\phi \vee \Diamond\neg\phi$. The meaning is: *either* ϕ *is true necessarily, or maybe* ϕ *is false.* To get something interesting, it is important to add some axioms and rules specific for the modality. In S_4 the additional axioms are:

(1) $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$,
(2) $\Box\phi \to \Box\Box\phi$,
(3) $\Box\phi \to \phi$,

and an additional rule, called *generalization*, is

from ϕ, *derive* $\Box\phi$.

Note that there are formulas ϕ such that $\phi \to \Box\phi$ are not tautologies (for example, if ϕ is a propositional variable). This seems counterintuitive because of the rule of generalization. The reason is that in this logic we cannot derive an implication $\phi \to \psi$ by deriving ψ from the assumption ϕ.

14. *Kripke's "possible-world" semantics of modal and intuitionistic logics.* A crucial moment in the history of modal logics was the discovery of *possible world semantics*. The idea is very old, perhaps, this interpretation of modalities was known already to Aristotle, but it was only in the late 1950s when the American logician and philosopher S.A. Kripke (as a teenager) found a mathematically sound approach to possible world semantics, which is now called *Kripke semantics* [170]. The basic idea is that *true* should mean *true in a particular world*, while *necessarily true* should mean *true in all worlds*. In particular, this explains the generalization rule: if we derive a proposition without any additional assumptions, then it must always be true. This idea has to be developed further, since in general, formulas may have many occurrences of the modality and the occurrences can be nested, etc. Therefore, we not only need possible worlds, but also a relation between the worlds. It is a binary relation with some properties; we say that W_2 is an *alternative world* to W_1, if the relation holds between W_1 and W_2. Then one defines inductively the truth value of modal formulas. For propositional variables, we assume that they get a definite value true or false in each world; they can have different values in different worlds. If we already know the truth values of some formulas and we combine them by Boolean connectives, then the truth value of the compound formula is defined

in the usual way. The key step is when we know the truth value of ϕ for each world and we need the truth value of $\Box\psi$ in a world W_1. Then we define:

> $\Box\psi$ *is true in a world* W_1*, if* ϕ *is true in every world* W_2 *which is alternative to* W_1*; otherwise it is false.*

In order to avoid confusion with the usual concept of satisfiability, one often says that W *forces sentence* ϕ and writes $W \Vdash \phi$ when the sentence is true in the world W in the sense of possible-world semantics.

Example Suppose the relation on worlds is a linear ordering; we can think of it as one world in different times. Then $\Box\phi$ *is forced in a world* W_1 means that ϕ is forced in W_1 and in all future worlds.

Formally, in Kripke semantics one model is replaced with a set of models and a binary relation on this set. In the case of propositional logic that we consider here, single models are simply truth assignments to variables. The relation is called the *frame* of the model. The most interesting thing is that the properties of the frame determine the modal logic. In particular, reflexive and transitive relations determine S_4. What it precisely means is this.

Theorem 4 *A formula is derivable in* S_4 *if and only if it is forced in all models with reflexive and transitive frames.*

Notice that the link between the axioms and properties of frames is quite direct. The axiom schema (2) expresses transitivity and (3) expresses reflexivity. Other frames determine other systems of modal logics.

Example Suppose that ϕ is forced in a world W_1, but it is not forced in an alternative world. Then in W_1 $\Box\phi$ is false. This shows that $\phi \to \Box\phi$ is not a tautology.

Intuitionistic propositional logic can be interpreted in S_4 as follows. For an intuitionistic formula ϕ, we define its translation ϕ' into S_4, essentially, by putting boxes everywhere. More precisely, we define inductively: a propositional variable x is translated as $\Box x$; the translations of $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$ and $\phi \to \psi$ are respectively $\Box\neg\phi'$, $\Box(\phi' \vee \psi')$, $\Box(\phi' \wedge \psi')$ and $\Box(\phi' \to \psi')$, where ϕ', ψ' are the translation of ϕ, ψ.[21] Then a sentence ϕ is an intuitionistic tautology if and only if its translation ϕ' is an S_4 tautology.

We can check that the law of excluded middle is not valid in intuitionistic logic. Translating the formula $x \vee \neg x$, x a propositional variable, into S_4 we get $\Box x \vee \Box\neg x$. If this were true, then we would have either x true in all alternative worlds, or x false in all alternative worlds. But a model can easily be constructed such that in a world alternative to a fixed world W_1 x is true and in another it is not.

[21] Boxes in front of disjunctions and conjunctions can be omitted.

Notice that this interpretation in turn gives possible world semantics for intuitionistic propositional logic. It is, however, easy to define Kripke models for intuitionistic logic directly. For example, the defining clauses for the negation and the universal quantifier are:

$$\begin{array}{ll} M \Vdash \neg\phi & \text{if for no } N \text{ alternative to } M,\ N \Vdash \phi; \\ M \Vdash \forall x\ \phi(x) & \text{if for all } N \text{ alternative to } M \text{ and all } a \in N,\ N \Vdash \phi(a). \end{array}$$

In Kripke models of first-order intuitionistic sentences the worlds are structures with relations and functions for the corresponding symbols of the sentences. It is further assumed that if W_2 is an alternative world for W_1, then the structure W_1 is a substructure of W_2. The equality relation has to be interpreted as an equivalence relation because, in general, equality is not preserved in alternative worlds.

2.4 Programs and Computations

The history of computations is as old as mathematics. As a matter of fact, early mathematics was just the art of computation. When proving more and more general results, mathematicians gradually started to consider problems that were not computable practically, but computable only *in principle*, which means computable having unlimited time for computations. Eventually they arrived at problems that are not computable even in principle (but the proofs of such non-computability were found much later). The algorithmic aspect has been, however, always eminent. You cannot apply a theory to practical problems, if there is no way to compute the results. We believe that our interaction with the real world is of a computable nature, thus any theory should produce instructions for computations, otherwise we cannot test it. Of course, there are many theoretical results that do not lead to algorithms, for example, results showing the non-existence of certain structures, the impossibility of some computations, etc. (some of these will be treated in Chap. 4). There we do not expect any output of the form of an algorithm, they only serve as warnings: *do not try!* Others give us insights, explanations, etc., but these insights should eventually help us to solve very concrete problems.

In the second half of the 20th century the theory of computing, or rather its applied parts, split from mathematics. Computing has become a new type of industry. Like physics, computer science studies very concrete things: computers and computations with them. Yet there are many purely theoretical problems concerning computing, which rather belong to mathematics. The main problem is what can be computed with limited resources (resources being time, the size of computers and their memory). The reason why this is a fundamental question which is not relevant only for computer science is that our interaction with the world is subject to the same type of limitations. Our brains are some sort of computer too and we too have very limited time for using them as individuals but also as humankind. Because of that these questions are very relevant for the foundations of mathematics.

The main property of algorithms is that for each particular instance of the problem for which they are designed, shortly *for each input*, they should only use a *finite* amount of time and space. This is clearly a necessary condition, otherwise we cannot realize computations physically. For practical computations, this is surely not enough. There are problems that need so much time that they are practically not computable. This leads to the study of the amount of resources, namely time and space, needed for computations, which is the subject of the *computational complexity theory*. Before I consider such subtle questions about computations, I have to explain the general concept whose only limitations are the finiteness of resources used.

In the theory of computation there is the same duality that we observed in logic. There is a description of computation called *algorithm* or *program* and the actual run of the algorithm or program, which are mechanical or electronic operations performed by a person or a computer. There is a slight difference in the use of the words 'algorithm' and 'program'. An algorithm is usually a higher level description which is not quite formal; it is assumed that the details of what to do precisely will be filled in by people using it, or programmers who transform the algorithm into an actual program, the *implementation* of the algorithm. A program is, on the other hand, a precise definition (in a programming language) of what the computer should do. When we want to compute something, we have to divide the task into a sequence of elementary simple tasks. People not only know a lot of problems that can easily be computed, but also are able to fill in gaps, thus the "elementary" tasks need not to be so simple as those used by computers. Therefore, people only need an algorithm, while a computer needs a program.

In natural languages there is a special class of sentences that corresponds to programs, the imperative. Cooking recipes are also some kind of program, so are various instructions for use, etc. The real programs work with symbols and numbers; on the lowest level they always use only bits, which means two values. In mathematics children start with algorithms for addition and multiplication. Then they learn some algebra. An algebraic expression is an algorithm. In order to compute the value of an algebraic expression for particular numbers, we perform the operations using the algorithms for addition and multiplication as subroutines. But only some computations can be defined by algebraic expressions. One of the most ancient algorithms is attributed to Euclid; it is an algorithm for finding the greatest common divisor of two integers. For this problem, there is no algebraic expression as for many others. Incidentally, Euclid's algorithm is very efficient and still used in practice. The word '*algorithm*' is also old; it stems from the name of Al-Khwarizmi, an important mathematician of the 9th century Baghdad school.

What a computer computation is everybody knows, but let me quickly describe it anyway, in order to introduce a few basic concepts. A computation starts with input data. During the computation new internal data are created. I will call both just *data*; these are strings of symbols, numbers or bits. A program contains commands and tests. Commands specify simple operations that should be performed with data. Tests specify simple properties of the data. The computation proceeds according to the result of the tests. A part of the program can be skipped, or the computation returns to a command before (this is called *a loop*). The existence of loops is an important feature. A loop may occur because there is an instruction to return to a previous part of a program (the *go to* control statement) or there can be a *repeat* statement.

Consider a "practical program" for driving a nail into wood: *Keep* (repeat statement) *hitting the nail* (command) *until it does not stick out* (test). It consists of a simple loop where we hit the nail and check, if it still sticks out. The reason for the loop is that we cannot specify *a priori* how many times we have to hit the nail. This kind of construction enables programmers to describe something that they do not know how much time it will actually need. In such cases the programmer must have some reason to believe that the computation will eventually stop; for instance, the programmer may have an upper bound on how many times the computation will go over the loop in the worst case. (In our "nailing" program 100 seems to be a safe upper bound.)

The concept of an algorithm had been defined in mathematics more than a decade before electronic computers were built. It was not only an advance in electronics that led to the construction of the first computer, but also an advance in mathematics. Several definitions of what is computable were proposed of which I will mention the three most important ones, two mathematical definitions (Turing machines and recursive functions) and one more practical definition, in fact a whole class of definitions (programming languages).

Turing Machines

Alan Turing
Courtesy of King's College, Cambridge University

The first approach to defining computations that I want to mention here, though a mathematical one, is friendlier for non-mathematicians. This concept is due to the English mathematician Alan M. Turing (1912–1954) and it bears his name: *the Turing machine*. A similar concept was defined independently by the American logician (born in Poland) Emil L. Post (1897–1954). Both definitions appeared around 1936 [221, 293]. The description of a Turing machine looks as if it were a technical device, very much like a primitive computer. A Turing machine consists of a control device connected to an infinite tape. The tape has distinct squares that can hold a symbol from an alphabet. The machine can read the symbol in the square that is currently scanned by the head and possibly rewrite it to another symbol. The control device is a finite machine which has a finite number of states (in mathematical terms, it is a finite automaton). We do not care how the machine is constructed, we only require that the current state and the currently read symbol determine what the machine does, including the next state to which it switches. A Turing machine operates in discrete time intervals, like a real computer, and in each step it reads, possibly rewrites a square on the tape, moves the tape one square to the right or to the left and switches the state of the control device.

The amazing thing is that, however primitive and cumbersome it is, it can perform any algorithm! Of course, the input data must be suitably presented; for example, when computing with natural numbers, we have to use the binary or a similar representation. Once I had the opportunity to admire the collection of physical models of Turing machines of a German professor who was fascinated by this fact. As they said, every new assistant researcher in his department got as the first assignment the task of constructing one. The exposition rather than giving insight into the concept of Turing machines, nicely showed the development of electronic components. The first machines were built from mechanical telephone components, while the last one used modern electronics. Those were the old days, nobody would do it that way nowadays. The best way to visualize such a concept nowadays is on a computer screen. This is not only an entertaining experiment, but also shows that the concept of a Turing machine is not a technical one, it is as mathematical as other mathematical definitions of computation. Consider the following argument: if you can represent a Turing machine in a computer, then you can represent it as a mathematical structure as well.

A Turing machine is a very cumbersome device. Even if you only want to design it to do such simple tasks like the addition of two numbers, you need to do a lot of simple but tedious work. So why has this concept been chosen and why is it still being used in theoretical computer science? The reason is that the simpler definition we are able to find for a given concept, the more the definition will reveal about the essence of the concept that is defined.

I will now give an informal justification of the concept of the Turing machine. I will argue that computations on any physical device can be simulated by a Turing machine.

Firstly, such a device should be able to store the input data, auxiliary information obtained during the computation and it should be able to present the output data. These can be represented using various data structures, but the simple data structure are strings in a finite alphabet. This suggests the use of a *tape*. As we cannot *a priori* bound the size of memory needed during the computation and the size of the output, we need an infinite tape. It makes little difference if the tape is infinite in both directions or only in one.

Secondly, the changes of the data recorded on the tape have to be simple. So why not to change just one symbol at a time? The place where the change is done must be determined by a simple rule. This requirement alone does not lead directly to the concept of the head of the machine, as we can easily think of various natural rules how to determine where the change should be done. However, if we think more mechanistically, it does not seem natural to make a change at one place and then in a single step to jump to another distant place. Thus the most natural thing is to have a pointer that defines the place in the data structure that is to be processed and allow the pointer to move only one step per unit time.

Finally, the whole process of local changes in data has to be controlled somehow. Saying that we have a finite list of rules that the machine follows would correspond to the intuitive concept of an algorithm as used in mathematics. However, to get a picture of a physical device that performs computations independently of us, it is better to talk about a mechanical device that does the elementary steps.

Turing's original justification in his seminal paper "*On Computable Numbers*", from 1936, was different. The main difference is that he analyzed what algorithms can be done by a *human*, rather than talking about physical devices. He had two reasons for that. Firstly, the idea that the human brain is just a very complex physical device was not so widely accepted at that time. Secondly, the available computing machines were able to perform only very simple algorithms. Interestingly, he did talk about "computers", but the meaning of this word back then was "a person who computes". When he reached the conclusion that a computation of a "computer" can be simulated by his concept, he finished by saying: "*If this is so, we can construct a machine to write down the successive formulae, and hence to compute the required number.*" Hence, humans are no better than machines.

Programming Languages

The construction of first computers in the 1940s changed the situation dramatically. Before we needed to communicate algorithms only to people and computations were performed by people (with some help of calculators at the later stage). Thus it sufficed to describe an algorithm in a natural language, using, of course, some mathematical terms. The precise definitions, like the Turing machine, were needed for pure research, for instance, to show that there are problems that are not solvable using an algorithm. Computers, first of all, needed precise definitions of algorithms. They are not able to fill in gaps and use hints instead of a full description, nor are they able to understand our language full of vague words. But that was only part of the problem. What was needed was a communication means between a person and a computer, a language that *both* can understand and that is sufficiently efficient in order to be used practically.

The first programming languages were more tailored for computers. Computer time was expensive, compared to the price of the time of researchers experimenting with them. Also computers were worshiped by many as being capable of doing miracles. Gradually the roles interchanged and now, in most cases, the time of a programmer is more expensive. Using the machine code for programming was a short episode. Soon a lot of effort was exerted to make the task of writing a program as simple and as efficient as possible.

A programming language borrows words from a natural language (almost exclusively from English) in the same way as logic does. These are words such as *for, do, repeat, stop, go-to*, etc. Their number is small, but it is not because we want to use as few words as possible. Furthermore, one uses mathematical symbols, mainly for arithmetic, as some numerical computations occur in most programs. The languages also offer *libraries* of functions that are often used, so that programmers do not have to program the functions again and again that often occur in programs. For instance, JAVA includes a function for finding the greatest common divisor (whose implementation is very likely based on Euclid's algorithm), but it also contains much more difficult ones; for example, you can ask for a random prime number in an interval. Not only libraries of functions simplify programming, but also the types of constructions that are possible in a given language are very important. One of the most important recent inventions is the concept of an *object*. It documents very nicely the convergence to human languages. Formally, it allows the programmer to impose a certain hierarchical structure on the program. Practically it means that the programmer can use features of human thinking that are not based on logical or mathematical deduction but on the vague concept of similarity. The point is that similar *real* objects can be treated in the same way. Thus in object oriented programming we group similar function and we allow the same procedures to be applied to them. Such features, however important they may be for practical programming, are irrelevant from the point of view of theory. In order to be able to define all functions in a programming language, one needs a very tiny fraction of it; we can do with very elementary commands and operations, the more complex ones can be programmed from the simpler ones.

What is the most amazing fact on programming languages is how many people "speak" these languages. Using a programming language is not the same as knowing it; it is an ability similar to a good command of a natural language; it is the ability of *thinking* in terms of the programming language. But, unlike natural languages, programming languages are completely formal systems, which shows that people can use a completely formal language, if there is a strong incentive to do so. Obviously, these languages can only express algorithms, but that is already quite a rich domain.

We should also note that there are big differences between architectures of different languages. So we should consider each single programming language as a different model of computable functions. Another thing to note is that a program not only defines a computable function, but also determines to a large extent how the function is computed. However, this is a common property of all definitions of computable functions.

Noncomputable Functions

Our practical experience with computation suggests that there are easy functions and difficult ones. The easy ones are computable practically. For the difficult ones, the problem seems to be that we do not have enough time and memory to compute them, but, perhaps, they are computable in principle? In fact there are functions that are very difficult, but in principle computable, but there are also functions, that cannot be computed even with arbitrary resources available. The main example of such a function is the *halting problem*. The task is to compute whether a given program applied to given data will ever terminate. This is, actually, an important question because a computer that is not producing any output is considered to be stuck, no matter how ingenious the computations it may be performing inside. Given a program and data we can perform an experiment by running the program on the data. If it stops, we know the answer, but how long should we wait? In practice we wait at most a couple of minutes and if nothing changes on the screen we know something has gone wrong. It can be proved, however, that there is no way to estimate how long it is necessary to wait. Thus sometimes we have to interrupt the experiment without getting a definite answer. So this is not a way to find out whether a program stops on given input data. The result about the halting problem says even more: not only can we not solve the halting problem by trying to run the program for a limited amount of time, but there is no way at all to decide it in finite time. A curious fact is that even an apparently much more restricted version of the halting problem is algorithmically unsolvable: to decide if a program *running on its own code* will eventually stop.

Some programmers may be surprised by the fact that finite problems such as the halting problem are not algorithmically solvable because these problems almost never occur in their daily practice. The halting problem is not computable because there is no bound on for how long a time the program may run. In practical problems there are usually some implicit bounds on how big the number tested should be, for how long, etc. The halting problem is a decision problem, thus it can be represented as a function with two possible outputs YES and NO (if we want to have mathematical objects, we can use 1 and 0 instead). The above observation about bounding the running time suggests the following noncomputable function f, which I will call *the halting bound*. For a natural number n, we define $f(n)$ to be the maximum of all running times of programs of length at most n applied to their own code.[22] We do not consider the programs that do not stop when applied to their own code. If f were computable, we could solve the halting problem as follows. Given a program P of length n, we would first compute the bound $f(n)$ and then we would run the program on its code for $f(n)$ steps. If P stops, then we are done, we know it stops. If, however, if does not stop within the time limit, we know nonetheless that it will never stop. So again we can give a correct answer. What is interesting in this example is that we actually do not need the precise value of $f(n)$. The same argument works for any function g such that $g(x) \geq f(x)$. Hence the reason why the halting bound is not computable is not that the values of it have a special property, but it is the growth of the function. The halting bound function increases so rapidly that there is no program that can compute such a function.

The halting bound is closely related to the well-known *busy beaver function.* This function, denoted by $\Sigma(n)$, is defined as the maximal number of steps that a Turing machine with n states and an empty tape can make and halt. Notice that we take the maximum over all n state machines that *halt*, which is a condition that we cannot algorithmically test. Further, we assume that the machine uses two-letter alphabet on the tape. This function is also noncomputable.

Some of the nicest examples of algorithmically unsolvable problems come from number theory. Given an equation, it cannot be decided by an algorithm if the equation has a solution in the domain of natural numbers. Recall that equations with integer coefficients are called Diophantine equations, and we are interested only in integral solutions of these equations. I gave examples of such equations in Chap. 1, page 56. Let me mention one more important example, the Pell equation

$$x^2 - dy^2 = 1,$$

[22] For the sake of simplicity I restrict myself to the more specific version of the halting problem.

where x, y are unknowns and d is a given natural number. This equation has been studied and we know, for example, that it has a solution for every d that is not a square. There are several other classes of equations that are relatively well understood (in particular some classes of quadratic equations), but there is no general theory of all Diophantine equations. In fact already the work of Diophantus was criticized for giving many ingenious proofs, but no general theory. The algorithmic unsolvability of Diophantine equations gives an explanation why he could not do it. This result does not exclude that in the future we will have a nice general theory, but if this ever happens, it will not be the kind of theory that gives us formulas or algorithms for solutions. I will show explicit examples of algorithmically undecidable Diophantine equations in Chap. 4.

In logic there are several problems that are not computable. The most basic one concerns first-order logic; for a given sentence ϕ it is undecidable by an algorithm whether or not ϕ is logically valid (= provable). By the same argument as for the halting problem, it implies that we also cannot bound the lengths of the shortest proofs of valid sentences. This gives an intuitive explanation, why provability in first-order logic is hard. I will say more about it shortly.

Universal Machines

Of the many other important results in computation theory I will mention only one. It says that there are in some sense *universal machines* or *universal programs*. This means that there is a program U such that for any program P and input data x, if we use x and the program P as input data for U, the universal program U will produce the same output as P produces on x, assuming P stops and produces an output. We say that, given an input (x, P), the machine U *simulates* the computation of P on the input x. This looks like a nice statement for mathematical recreation or for impressing philosophers, but, in fact, it has very important practical consequences. Suppose you use real Turing machines for computations, then the result says that you need only one. It is much easier to write additional data on the tape than to construct a new machine for each new task. This is also what computers do: a computer uses one universal program, called the operation system. When you run a concrete program, you give a file with the program to the computer and the operation system simulates your program. The same phenomenon can be observed on a higher level: a compiler (a program that enables you to run your programs on a computer), say, for the programming language C, can be written in C. Such a compiler is a universal C program.

It is important to stress that a universal program U behaves in the same way as the given simulated program P also when P does not stop, which means U also does not stop. In order to be able to define a universal program, we have to consider all programs, not only those that always stop. Hence, U will not stop on some inputs. We cannot require U to stop on every input because then U would solve the halting problem.

In order to prove that, say, a universal Turing machines exist, one defines such a machine explicitly. This is the best way to prove it and the constructed machine is not terribly complicated, but it is rather boring to write down the description of such a machine and it is even more tedious to check that it does what it should. To write down a description of such a machine with no explanations is a rather cheap trick to impress readers. So instead, I will use a simple explanation why such a universal machine should exist, an explanation based on the assumption that Turing machines can compute any computable function. Suppose your task is, for a given Turing machine, input data and a number n, to find out what the machine will do with these data within n steps of computation. This is, clearly, something that you can solve algorithmically: you simply plug in the input into the machine, switch it on and see what happens. Therefore, this should be a computable function. Let me spell it out:

> *The function that for a given description of a Turing machine M, an input string x and a number n, gives the result of the n steps of the computation of the Turing machine M on the input x is a computable function.*

Hence this function is also computable on a Turing machine. So take a Turing machine that computes this function and modify it so that instead of using a fixed number n, it successively tries $n = 1, 2, \ldots$ until the simulated machine stops; if the simulated machine does not stop, we let it run forever. The resulting machine is a universal Turing machine.

We can turn this argument around and say that having a universal function is a natural requirement for any class of functions that we would like to call computable because the action of executing a given program on given data should be computable.[23]

Considered from a broader point of view, universality is also an attribute of intelligence. Knowing a solution of a practical problem, we can instruct another person how to do it (provided we speak the same language). Dogs are quite intelligent, but they are able to interpret only single commands. Still universality is present in the brains of even less intelligent animals at least to some extent. Of course, we cannot communicate programs to them, but we can *train* them to perform quite complex tasks. What we train them to do is by far much simpler than what the animals need for their life in nature. Universality of the brain is the ability to learn to do things that the body is capable of doing. Of course, universality is not sufficient for a system to be considered intelligent; we do not consider our computers to be intelligent. But, perhaps, there is a kind of higher universality that is the essence of intelligence.

Universality sounds sublime, but it is not a luxury. Special purpose chips are used less and less, as mass production prefers universal ones. Most computers are used exclusively for word processing, yet it is not profitable to produce machines only for this purpose. Universality of the brain was the main selective advantage of *homo sapiens*. One may wonder how a brain shaped for a primitive hunter can play chess, do mathematics, program computers, etc. The explanation is that nature has discovered that a universal brain is much better if the conditions change unpredictably.

[23] This only concerns partial functions, functions that may be not always defined, see Notes.

Animals with universal brains can easily adopt a particular useful behavior without the long process of natural selection that is otherwise needed to "hardwire" it into the brain.

The Undecidability of First-Order Logic

Alonzo Church
Courtesy Princeton University Library[24]

Once a formalization of proofs in first-order logic was achieved, the next natural question was how difficult is to decide whether or not a given sentence is logically valid. In particular, is there an algorithm for deciding if a sentence is logically valid? This problem became famous as Hilbert's *Entscheidungsproblem*, which means *decision problem.* The importance of this problem stems from the fact that if the answer were positive, then it would be possible to automatize mathematics, at least in principle. The undecidability of first-order logic was proved by the American logician Alonzo Church (1903–1995) and independently by Turing; their papers [43, 293] appeared in 1936 and 1937, respectively.

Let us see how the decidability of first-order logic would help solve mathematical problems. Suppose that we are working in a theory that can be axiomatized by a finite list of axioms $\alpha_1, \alpha_2, \dots, \alpha_n$ and we want to find out whether or not a sentence ϕ is a theorem. Then ϕ is a theorem in the theory if and only if the sentence

$$\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n \to \phi,$$

(which expresses that the conjunction of axioms implies ϕ) is a logically valid sentence. Hence having an algorithm for the predicate calculus, we would also be able to solve the decision problem whether or not a sentence is a theorem in this theory.

Zermelo-Fraenkel Set Theory cannot be axiomatized by a finite set of axioms, but this does not matter. We will see that there is a finitely axiomatizable theory (Gödel-Bernays Set Theory) that proves exactly the same sentences about sets. Since Zermelo-Fraenkel Set Theory suffices for essentially all mathematics, a positive solution to the *Entscheidungsproblem* would enable us, in principle, to solve all mathematical problems mechanically.

This shows the theoretical importance of the problem, but for practice, it is not so essential. Note that although we do not have an algorithm for the *Entscheidungsproblem*, we can still do something: we can systematically generate all possible proofs and create an infinite list of theorems such that a sentence is a theorem if and only if it appears on the list. But, if are interested in a particular sentence which is not a theorem, we will wait forever and never learn the answer. So this does not

[24] Alonzo Church Papers. Manuscripts Division. Department of Rare Books and Special Collections. Princeton University Library.

give us a decision procedure. What is, however, more important is that generating theorems in this way is extremely inefficient. As researchers in artificial intelligence know, one can obtain only very simple theorems in this way, thus such a method of generating theorems is practically useless. But the same could have been the case if it had turned out that first-order logic was decidable—a decision algorithm for first order logic would have been of no practical use either. The bottom line is that, from the practical point of view, it is not important whether there is an algorithm for the decision problem, but whether there is an *efficient* algorithm. (All of Chap. 5 will be devoted to the concept of efficiency of computations.)

The undecidability concerns pure first-order logic. If we add some axioms, the set of provable sentences may be decidable. For example, if the axioms are inconsistent, all sentences are provable, thus the decision problem is trivial. There are, however, also nontrivial examples. Recall that there are complete axiomatizations of the natural numbers with $+$ as the only operation, and of the real numbers with the operations $+$, $\times$ and the relation $<$. Both theories are also decidable.

It is not just a fluke that completeness and decidability appears at the same time in these examples. Any complete theory is decidable. This is an easy fact based on the procedure that generates all proofs in a theory. Recall that if a sentence ϕ is provable, then eventually the procedure will find the proof and thus certify that ϕ is a theorem. But if T is complete, we know that if ϕ is unprovable, then $\neg\phi$ is provable. So we always find in finite time either a proof of ϕ (when ϕ is a theorem) or a proof of $\neg\phi$ (when ϕ is not a theorem).

Notice that this only gives us an algorithm without any estimate on its running time. Considering how inefficient is to generate all proofs, one may suspect that the decision algorithms for decidable theories could also be very inefficient. Indeed, this is the case; for the two problems above, the complexity is doubly exponential and exponential, respectively, and for some other decidable theories the lower bounds are even higher.

There are also undecidable theories. first-order logic, as a theory with no axioms, is an example. More interesting examples are Peano Arithmetic and Zermelo-Fraenkel Set Theory. In fact, any theory in which one can prove some basic propositions about numbers is undecidable. Also, the proofs of the undecidability of the predicate calculus are based on proving that some finitely axiomatized theory is undecidable. There will be more about it in Chap. 4.

Recursive Functions

Here is a purely mathematical definition of computability. The concept of a *recursive function*[25] is due to the American logician Stephen C. Kleene (1909–1994) [153]. This approach is very mathematical, as we only consider computations with natural numbers and define the class of functions that can be computed, the re-

[25] a.k.a. *general recursive function*

cursive functions. By functions we mean functions of one or more variables, hence, what we usually call arithmetical *operations* are also functions. Arithmetical functions are not only important examples of possibly computable objects, but also they have a sufficiently rich structure. Thus it is possible to encode other computations only using operations on natural numbers. In general we have to choose a suitable domain for data. In the case of Turing machines the domain consists of sequences of symbols, while the domain of recursive functions is the set of all natural numbers.

In logic it is very common to define a class inductively by saying that some initial elements belong to the class and giving some operators that produce new elements in the class. The class consists of those elements that can be produced in this way. (Think of, say, proofs as defined by axioms and derivation rules.) Kleene, as a logician, used this approach. He took some basic functions and considered several operators producing new functions from given ones. The basic functions are quite simple, such as the constant function 0, the successor function $x + 1$, etc., so they clearly should be considered computable. We can add other functions, for example, addition and multiplication, that we surely consider computable, but it is not necessary as the operators enable us to produce them from the basic ones. The operators are also simple. In particular, we take the operator of *composition* (also called *substitution*) of functions. Having two functions f and g of one variable, we may first apply f to the input number and then apply g to the value produced by f. The resulting function is the composition of f and g. We may also compose binary functions. Thus we can get, for example, the function $x + yz$ from addition and multiplication.

If we started with the basic arithmetic operations (addition, multiplication and constants) and only used composition we would only obtain the functions that can be expressed by algebraic terms (polynomials). Therefore, we need more operators. Another basic operator is the operator of *recursion*. The name '*recursive functions*' comes from this operator, but it is misleading, as this operator is still too weak to produce all computable functions. A special case of it is the operator of *iteration*, by which we compose a function with itself a given number of times. The most powerful operator is the *minimization* operator. It allows us to search for the smallest number satisfying a condition.

We can imagine recursive functions as follows. We expand our "algebraic" language by taking more operators on top of the composition. Having a sufficiently powerful set of basic functions and operators enables us to define all computable functions by an expression in this language.

A formal definition of recursive functions is in Notes.

The Church-Turing Thesis

Having definitions of computable functions the next natural question is how good these definitions are. What seems clear is that each of the definitions only describes functions that can be (at least in principle) computed. But do these concepts (Turing machines, programs, recursive functions, etc.) cover all computable functions? This is the same kind of a property that we studied in logical calculi and called completeness, but here we have a problem: we do not have a class of functions that we would consider as the computable functions and that would be defined independently of a concrete computational model. We do not have a purely semantical definition of computable functions. We do have a fairly clear idea about computable functions and all the computation models are in good agreement with it, (the idea is that a computation should use a finite number of elementary operations), but to make this idea precise we have to opt for one of the computation models. Before we choose one, it is, surely, worthwhile to compare them. In particular, is the class of numeric functions computable by Turing machines the same as the class of recursive functions? I have already said that all algorithms can be done on Turing machines, so it should not come as a surprise that the two classes coincide. In fact, if you think about these concepts, after a while you will realize that they are not so different as they appear at first glance. If you try to implement algorithms on a Turing machine you will soon realize that a programming language for it would be handy. That is exactly like asking for a higher level language instead of a machine code for your computer. So isn't a Turing machine rather a primitive programming language? In some sense it surely is. It is a programming language with a single data structure which is a linear array and a single pointer whose position can be incremented or decremented only by one. If you analyze it more, you may find an even closer correspondence, for example, the program lines correspond to the state of the control in the Turing machine, etc.

What about recursive functions? Also this definition can be interpreted as a kind of a programming language. It is a programming language for computations with natural numbers. It has some simple functions as primitive concepts, as most programming languages do. Then it has certain operators that we can interpret as possible constructions that can be used in a program. One of these is *composition*, that is used to form terms; this is also available in most programming languages. Another is *recursion*, again this construction is very common in programming languages. It helps when writing very short programs, but professional programmers try to avoid it whenever possible, as it does not give them good control of the computation. *Minimization* is not present in programming languages as a basic construct (it gives even less control of what is going on during the computation), but it can be programmed easily. On the other hand, *iteration* corresponds to a simple loop in a program and this is the most frequent construction.

Such mutual interpretations were found not only for the aforementioned three concepts, but for all that had been proposed. This is not a proof, but sufficiently good evidence that the concepts have been chosen correctly. The claim that these concepts characterize computable functions is called the *Church-Turing Thesis* (although neither Church nor Turing stated the thesis precisely in the way it is presented nowadays).

Can the Church-Turing Thesis be proved or disproved? Firstly, it cannot be proved or disproved as a *mathematical* statement because it is not a mathematical statement. It relates mathematical concepts to part of our practical experience for which we do not have a rigorous definition. Theoretically, it is possible that somebody might come up with a programming trick, an operator on functions, etc. that we would like to call computable but that would not be covered by the current definitions of computations. In such a case we would have a reason to abandon the thesis, but strictly speaking this would not be disproving. Considering the years of experience with programming, such a possibility is almost excluded.

The only way to make this thesis a little more formal statement is to interpret it as a postulate in *physics*. It perfectly makes sense to take the current physical theories and look whether the phenomena there have a computable nature and how they can be used for computations. The conjecture that all *physically* computable functions are computable on Turing machines, or their equivalents, is called the *Physical Church-Turing Thesis*. This question has been studied and lead to the new concept of quantum computing. Quantum Turing machines are a new important concept and it seems that they can solve some problems faster than classical Turing machines (as we will see in Chap. 5), but when computational complexity is ignored, the two models are equivalent. This is not a proof, but a strong evidence that the Physical Church-Turing Thesis cannot be disproved using quantum theory.

Naturally, also general relativity was used in the attempts to refute the Physical Church-Turing Thesis. There the situation is less clear. There are enthusiasts who believe that some likely occurring phenomena, such as black holes, can be used to solve problems that are not computable on Turing machines, others are more sceptical. In any case, this is only a theoretical discussion; nobody believes that such schemes can ever be used to help people compute. This is in contrast with quantum computing, where several teams of experimental physicists are working on constructing quantum computers. Nevertheless, the research into relativistic computations is extremely important because it concerns the fundamental question of what can physically be computed. (For more about it, see Notes.)

It has also been suggested that a noncomputable function could be encoded in fundamental physical constants. For example, the decimal digits of some constant could define a noncomputable set. If there also were a way to measure the value of the constant with arbitrary high precision, we would obtain a refutation of the Physical Church-Turing Thesis. In the most optimistic scenario we would also know that the digits of the constant encode a particular noncomputable set and we could use it to decide Church-Turing undecidable problems.

It should be noted that the Physical Church-Turing Thesis is meaningful only if the universe is infinite, which we still do not know. If space and time were finite, then there could only be finite computations. In such a case the fundamental role of computability would be replaced by the role of computational complexity. But I believe that computational complexity is equally important for physics even if the universe is infinite.

The Syntax and the Semantics of Computations

The distinction between syntax and semantics is essentially the same as in logic. Syntax is (descriptions of) Turing machines and programs in programming languages. In the case of recursive functions it is a little bit more subtle. There the syntax is a definition of a function in terms of basic functions and operators. Using a suitable notation for operators we can write these definitions as algebraic terms.

Semantics is given simply by the corresponding functions. We assume that a program starts on input data, computes, and then outputs the result of the computation and stops. The function is the assignment of the output data to the input data. Recall that in mathematics we treat function in the purely extensional way: we think of a function as a set of pairs *input-output*. When defining recursive functions, the functions are part of the definition, so it is even simpler (no wonder, it is a mathematical approach to computability). A Turing machine determines a function on strings in the given alphabet. We assume that, except for a finite segment, the tape contains blank squares. When the machine starts, the input to the machine is the word on the non-blank squares; similarly, when the machine stops, the output is what is written on the non-blank segment of the tape. Thus the machine always works with finite strings.

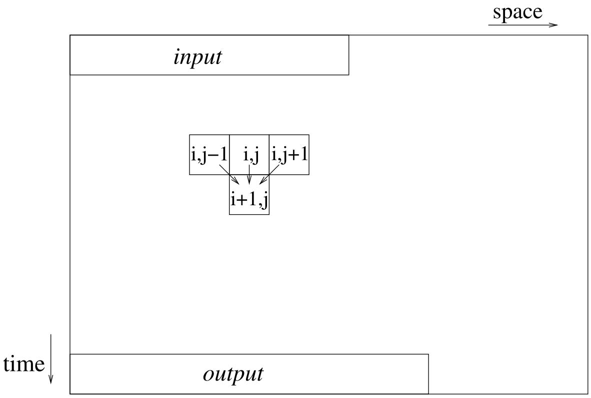
In logic we have not only formulas and models, but also *proofs*. A similar concept in the theory of computations are *computations*. When we are only interested in the question whether or not a function is computable, we do not have to define computations, instead we can use the definition of a computable function. But if we want to study practical computations, it is important to have a definition of a computation. In particular, time and memory requirements are of great concern for difficult problems, and those can only be studied using the concept of a computation. Mathematically, a computation is a sequence of elementary steps. What is a single elementary step depends on the model. For Turing machines, it is scanning the current square, printing a new symbol on it, moving the head to an adjacent square and changing the state of the control.

Furthermore, one needs the concept of computation for a single purpose program. Suppose we need to solve a concrete problem, but the program that we write probably will never be used again. Then the semantics based on functions does not make any sense, as we only need one output. The distinction between computable and noncomputable functions can only be made if we have *infinitely many* inputs. Computability only concerns the distinction between finite and infinite, namely, computable means that the process always ends after finitely many steps.

The Matrix Model of Computations

I am going to introduce yet another model of computations. My aim is to capture the combinatorial character of a computation. This model is useful for analyzing and proving theorems about computations, in particular, it can also be used for studying finite function, which can be done using the classical models too, but in a more cumbersome way. Furthermore, the main parameters of the model, the dimensions of the matrix, correspond to the main complexity measures, time and space, which also suggests a relation to physics.

Fig. 2.4 The matrix model



The main twist is to focus on *computations* instead of devices that do computations. I define a computation to be a matrix M with entries from a finite alphabet satisfying the following condition. There is a rule R that uniquely determines the entry $m_{i+1,j}$ (the entry in row $i+1$ and column j) from the entries that are in the row above (the row i) that are adjacent to it, see Fig. 2.4. Thus the entries $m_{i,j-1}, m_{i,j}, m_{i,j+1}$ determine the entry $m_{i+1,j}$, provided j is not the index of the first or the last column; if it is, then it depends only on two entries above (on $m_{i,j}, m_{i,j+1}$ if j is the first column, that is, $j = 1$, and on $m_{i,j-1}, m_{i,j}$ if j is the last column). Let me stress that the rule is uniform for all entries; it does not depend on the position in the matrix, except that it is sensitive to the sides of the matrix.

The computational interpretation of this model is as follows. We put input data on the first row. Then fill in the matrix row by row and read the output data in the last row. So the idea that we use to solve a particular problem, the algorithm, is hidden in the rule R. The advantage of viewing a computation in this way is that we eliminate the dynamical aspect of the computation by representing it by a fixed structure. A column j describes the content of the memory location j during the computation, the entries in a row i encode the content of memory location in time i. Thus the physical interpretation of the two dimensions of the matrix is time and space. The matrix is the trajectory of a computation. Furthermore, the rule that determines the system is local, as it should be in physics. Filling in the matrix can be viewed as a discrete version of a classical problem of solving an ordinary differential equation with a boundary condition; the boundary condition is the input data. An example of computation is given in Fig. 2.5.

In order to implement all algorithms in this model, we must allow arbitrarily large alphabets (because in a finite alphabet we have only a finite number of possible rules R). The letters in the alphabet will be used not only to code the bits in the memory, but also the currently implemented instruction from the program. We can avoid this problem by relaxing the definition a little. If we allow rules that determine the entry not only by at most three entries above, but at most k entries in the row above (adjacent to the entry right above), then we can do with a two letter alphabet (0 and 1). Various other generalizations are possible, such as using more dimensions

2	3	0	1	0	3	1	2
2	0	3	0	1	1	3	2
0	2	0	3	1	1	2	3
0	0	2	1	3	1	2	3
0	0	1	2	1	3	2	3
0	0	1	1	2	2	3	3

Fig. 2.5 The matrix of a computation that sorts the sequence 23010312 (the top row) into 00112233 (the bottom row). The computation is done by swapping every two consecutive numbers when the first is bigger than the second (this is the rule R). Thus each entry of the matrix, except for the first row, is uniquely determined by the two or three neighbors above it. This example is contrived, since there are no ambiguities as to which elements should be swapped. (Such an ambiguity would appear, for example, in 321, in which we can swap either 3 and 2, or 2 and 1.) In order to be able to sort any sequence we would have to use a rule that would decide which of the possible conflicting replacements should be done. Furthermore, this is not a very efficient way to sort. Fast sorting algorithms are based on a special choice of which pairs are compared and swapped; then one has to consider also pairs of non-consecutive elements

for space. But such improvements bring only insignificant gain in the efficiency of computations.

We should note that often more space is needed than is allocated for input data. In fact, we often want to keep the input data intact during the whole computation and use other memory locations for the data created by the computation. Thus we allow the use matrices with more columns than is the length of the input string. We put the input data at the beginning of the first row and fill in the rest uniformly by a symbol not present in the data (the 'blank' symbol). Then the number of columns is the size of memory used in the computation—the space complexity.

I will use this model for explaining computational complexity in Chap. 5.

Beyond Computability—Complexity Hierarchies

Naturally we wonder what is beyond the range of computable functions. Because things become more difficult, I will restrict myself to decision problems and I will only speak about sets of natural numbers. Formally, a decision problem is simply a set; the problem is, for a given element, to decide if it belongs to the set. We say that the decision problem is *algorithmically solvable*, or that the set is *recursive*, or *decidable* if there is an algorithm to decide this problem. Equivalently, it means that the characteristic function of the set is computable.

For example, let P be a program and let the problem be to decide for a given input, if the program will terminate on the input. Thus we are interested in the set of inputs on which P terminates. We know that this set is not decidable for some programs P (in particular for the universal program). Nevertheless, this set is not very far from computable ones. It can be defined using a single existential quantifier. It is the set of inputs x such that *there exists* a terminating computation on x, which we can represent as a string of symbols y. Notice, that the only noncomputable thing

$$\begin{tikzcd}
\vdots & & \vdots \\
\Sigma_3 & & \Pi_3 \\
& \Delta_3 \arrow[ul] \arrow[ur] & \\
\Sigma_2 \arrow[ur] & & \Pi_2 \arrow[ul] \\
& \Delta_2 \arrow[ul] \arrow[ur] & \\
\Sigma_1 \arrow[ur] & & \Pi_1 \arrow[ul] \\
& \Delta_1 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Fig. 2.6 The diagram of the Arithmetical Hierarchy. Δ_1 is the class of recursive sets, $\Delta_n = \Sigma_n \cap \Pi_n$, for $n = 1, 2, \ldots$

is how to get the string y; once we have it, we can verify that it has the required property. Another such set is the set of arithmetical equations that have a solution. Again, having a solution we can check by simple computation that it satisfies the equation, the problem is only how to find it. A very important example is the set of logically valid sentences of first-order logic. This set can be defined, by the completeness theorem, as the set of provable sentences. The latter of the two definitions is based on one existential quantifier: the set all sentences ϕ such that *there exists* a proof of ϕ.

Formally the relations with one existential quantifier can be defined by a formula of the form

$$\exists y\ \phi(x, y)$$

with ϕ denoting a computable binary relation. They are called Σ_1 or *recursively enumerable* sets. The Σ_1 stands for a single existential quantifier. The name '*recursively enumerable*' stems from the property of such sets that their elements can be enumerated by a recursive function. We can similarly define the class of sets definable using a single universal quantifier. This class is denoted by Π_1. As the negation of the existential quantifier is the universal quantifier, Π_1 is the set of the complements of sets in Σ_1. The classes Σ_1 and Π_1 are incomparable; there are sets that belong to Σ_1 but not to Π_1 and *vice versa*. Then we can use two quantifiers. Taking two quantifiers of the same kind does not produce new sets, but if we take two different ones, we get more complex sets. Thus we get the class Σ_2 by using formulas of the form $\exists y \forall z\ \phi(x, y, z)$ and class Π_2 by using formulas of the form $\forall y \exists z\ \phi(x, y, z)$. In general the class Σ_n (respectively Π_n) is the class of sets definable by formulas with n alternating quantifiers starting with $\exists$ (respectively $\forall$). A diagram showing relations between the classes is in Fig. 2.6.

One can show that at each step we get more sets, thus this hierarchy is increasing (but the pairs Σ_n and Π_n are incomparable). Hence using more *alternations* of quantifiers we can define more. This has an important consequence for logic. It proves that in general we cannot reduce the number of alternations of quantifiers in a formula.

It seems that we have exhausted logical means of defining sets, but that is not quite true. It is true that using only first-order formulas we cannot define more sets of natural numbers in the structure $(\mathbb{N}; 0, 1, +, \cdot)$. The sets of numbers definable by arithmetical formulas are called *arithmetical*, they are sets that belong to classes of the *arithmetical hierarchy* $\Sigma_1, \Pi_1, \Sigma_2, \Pi_2, \dots$.[26] But using higher order languages we can define more sets; we can define sets that are outside the arithmetical hierarchy. In higher order languages we can talk not only about numbers, but also about sets of numbers, sets of sets of numbers, etc. Let us look at the simplest example. Consider a formula of the form: *there exists a set* Y *such that* $\phi(x, Y)$ *holds true*. Formally, it is written as

$$\exists Y\ \phi(x, Y).$$

We use the capital letter Y to distinguish sets from numbers. We would like to use this formula to define the set of all x that satisfy it. It is, of course, important to specify what relations we should allow for ϕ because otherwise we could trivially define any set of numbers. When defining sets in the Arithmetical Hierarchy, we used formulas ϕ that were recursive relations. But now the formula speaks not only about numbers, but also about sets. Therefore, we allow ϕ to be any formula constructed from recursive relations and relations of the form $z \in Y$ with arbitrary first order quantifiers, by which I mean quantifiers binding numbers. (The first-order quantifiers are much weaker than a single second-order quantifier, so in fact, it would suffice only to use one universal quantifier for numbers.) Though Y may stand for arbitrary sets, a formula such as $\exists Y \phi(x, Y)$ cannot define arbitrary sets. Notice that Y is quantified in the formula, it is not a parameter, therefore it is as if we only used generic properties of all sets, not a particular set. Hence again, we only get some subset of all sets of numbers. However, we get more than we have in the Arithmetical Hierarchy.

This is only the first step after the Arithmetical Hierarchy. Then we can use more alternations of the second-order quantifiers—the quantifiers for sets, then third order quantifiers—quantifiers for sets of sets, etc. At each step of this construction we get more and more definable sets. After exhausting all levels of higher order languages we can invent a new way of defining sets again and go on as far as our imagination will allow. We will always get more and more sets. The process can never stop because the number of formulas will always be only countable, while the number of all subsets of natural numbers is uncountable.

Thus we can draw a crucial conclusion: even if we are only studying numbers, we may need higher order concepts, namely sets, sets of sets, etc.

Notes

1. *Total and partial recursive functions.* To make it simpler I have concealed an important distinction, the distinctions between partial and total functions. A *partial*

[26] These symbols are often used with the superscript 0: $\Sigma^0_1, \Pi^0_1, \Sigma^0_2, \Pi^0_2, \dots$.

function is like a function, but it is not defined for all arguments. We say that a function is *total* to stress that it is not partial. When we have a program that does not stop on all inputs it defines only a partial function. Formally, we can easily extend a partial function f to a total one, but this does not solve the problem. The problem is that it is not algorithmically decidable whether or not a program stops on a given input (the halting problem). So while we can easily redefine the function, we cannot rewrite a program P to P' so that P' prints the symbol ∞ when P does not stop. It is also not possible to change the syntax of the programming language so that only programs that always terminate can be written without essentially reducing the class of functions that can be programmed. Furthermore, partial functions are interesting things *per se*, therefore we should not prohibit them. In particular, a universal program defines a partial function that cannot be completed to a total computable function.

When talking about a total function, we say *a computable function* or *a recursive function* meaning the same (if we talk about numeric function); if it is a partial function we call it a *partial recursive function*.

It is essential to use partial functions when defining universal functions because of the following:

Theorem 5 *The class of total recursive functions does not have a universal function.*

This can be proved very easily. Suppose $U(x, y)$ is a universal function for total recursive functions of one variable. We fist transform U to a univariate function u by defining $u((x, y)) = U(x, y)$, where (x, y) denotes a pairing function, (say $(x, y) = ((x + y)^2 + 3x + y)/2$). Thus for every unary recursive function f, there exists a number n such that for all m, $f(m) = u((n, m))$. We claim that u cannot be recursive. If it were, then v, defined by $v(x) = u((x, x)) + 1$, would also be recursive. But then, for some n, $v(n) = u((n, n))$, which is a contradiction because $v(n) = u((n, n)) + 1$.

Note that although the theorem explicitly talks about recursive functions, it concerns total computable functions in any model. Thus

- there exists no Turing machine U that is universal for Turing machines that halt on every input and such that U also halts on every input,
- there exists no programming language in which all total computable functions can be defined and only such functions,

and so on. The proof of the theorem is an application of *Cantor's diagonal method*, which we will encounter again in the sequel.

2. *Definition of recursive functions.* The set of recursive functions is the minimal set of functions defined on the natural numbers that satisfies the following four conditions:

   a. it contains the following functions: the successor function $S(x) = x + 1$, the identity function $I(x) = x$, the projection function $I(x, y) = y$ and the constant zero (function) 0;

b. if $f(x_1, \dots, x_n)$ and $g(y_1, \dots, y_m)$ are recursive functions (the sets of variables of these functions do not have to be disjoint), $1 \le i \le n$, then

$$f\big(x_1, \dots, x_{i-1}, g(y_1, \dots, y_m), x_{i+1}, \dots, x_n\big)$$

is also recursive;

c. if $f(y)$ and $g(x, y, z)$ are recursive, then the function $h(x, y)$ that satisfies (is defined by)

$$h(0, y) = f(y),$$
$$h\big(S(x), y\big) = g\big(x, y, h(x, y)\big)$$

for every x, y, z, is also recursive;

d. if $f(x, y)$ is recursive and satisfies the condition that *for every x there exists y such that* $f(x, y) = 0$, and $h(n)$ is defined to be the least m such that $f(n, m) = 0$, then h is also recursive.

Thus the set of recursive functions is obtained from initial functions of (1) by applying operators of composition (2), recursion (3) and minimization (4). Note that the condition in (4) cannot be effectively tested, we can only *prove* it using some set-theoretical assumptions. To get an effective definition we have to consider the larger class of *partial* recursive functions; then we do not have to test the condition in (4). Notice that minimization corresponds to loops in the program, such as `do ... while ...`.

The name '*recursive*' probably originated by mistake. Gödel introduced a class of numeric functions only using 1., 2. and 3. hoping that it would contain all computable functions. He was shortly corrected by Herbrand, who pointed out that 4. is also needed. Among these three the most important was 3., the operator of recursion—therefore recursive functions. The class defined by 1.–3. is a proper subclass of all recursive functions, called *primitive recursive functions* nowadays.

3. *Definition of Turing machines.* A Turing machine is determined by the tape alphabet and the control device. The control device is simply a finite automaton. Thus, formally, a Turing machine is a structure (Σ, Q, τ), where

$$\tau : \Sigma \times Q \to \Sigma \times Q \times \{L, R\}.$$

The set Σ is the *tape alphabet*, Q is the set of *states* of the finite automaton. Given a symbol $a \in \Sigma$ and a state $q \in Q$ the value of the *transition function* $\tau(a, q) = (b, r, d)$ determines the symbol b printed by the machine, the new state r and the direction of the movement of the head d.

A *configuration* of the machine is determined by (1) the content of the tape, which is represented by an infinite sequence indexed by integers, (2) the state of the finite automaton, which is $q \in Q$, and (3) the position of the head, which is an integer.

To define a computation of the machine, one has to define one step *transition* from one configuration to the next one. This is easy, provided that we have in mind the interpretation of the concepts. Then a computation is a sequence of configurations where the consecutive configurations are obtained by single transitions.

In order to use such a machine as an algorithm, we have to make several provisos. Firstly, we want only to process finite sequences of symbols (these are called *words*). Thus we define an initial configuration to contain such a word on a specified part of the tape with the ends of the word marked and the rest of the tape being uniformly filled with the same symbol. The most convenient way to do this is to reserve a special symbol of Σ as the blank symbol and require that the input and output words do not use it. Then we have to fix an initial state of the automaton, a final state of the automaton and the initial position of the head, say the index 0. The output is the maximal length word of non-blank symbols containing the 0 position that appears at the moment when the automaton reaches the final state. Thus the machine determines a partial mapping on the set of all words of non-blank symbols.

4. *The matrix model of computation—continued.* Let Σ be the alphabet of symbols used in the matrix. To define this model formally, we have to solve the problem with the boundaries. The standard solution is to add columns on both sides filled in with an extra symbol, say, #. Then we can say that each entry inside is determined by the three entries above. Hence the model is given by an alphabet Σ, a rule $\rho : \Sigma' \times \Sigma \times \Sigma' \to \Sigma$, where $\Sigma' = \Sigma \cup \{\#\}$, and the two dimensions of the matrix.

   A computation of a Turing machine can be represented by such a matrix, except that we may need infinitely many columns and rows. In the first approximation we take simply the tapes with their content in the order of the computation of the machine. This is not quite correct as the content of the next tape depends also on the state of the control and the position of the head. This can be fixed by taking a larger alphabet in which we can encode also the state of the control. Using the notation for a Turing machine above, we take as the larger alphabet the disjoint union of Σ and $\Sigma \times Q$. Then an entry $a \in \Sigma$ in the matrix means that a is on the tape and the head is elsewhere, and $(a, q) \in \Sigma \times Q$ means that a is on the tape, the head is currently on the same square and the control is in state q. Thus rows encode configurations of the machine. We can make the matrix finite, if we consider only computations on inputs of lengths up to a fixed number n and assume that all computations terminate. Then the machine uses only a finite portion of the tape and we can bound the number of steps.

   Other combinatorial models are related to the matrix model, in particular arrays of automata and Boolean circuits. A *Boolean circuit* consists of gates and connections between them, called wires. It has input gates, inner gates and output gates. The role of input and output gates is clear; the inner gates process information that reaches them and send it further by wires. '*Boolean*' means that information comes in bits, denoted usually by 0 and 1. In practical circuits feed-back is very important, but in theoretical studies we mostly use circuits without any feed-back, just to make the model simpler. A matrix model with the alphabet $\{0, 1\}$ is a Boolean circuit (the first row being the input gates, the last row the output gates, etc.). The difference between the two models is not big. As I have noted, we can restrict the alphabet to {0, 1} by allowing dependence on a little more distant places in the matrix. The restriction to a rectangular grid is also inessential. When it is needed to send a bit to a more distant place, we can do it in several steps.

5. *Relativistic computations.* Relativistic computations were proposed independently by I. Németi, D. Malament, I. Pitowski and M.L. Hogarth in the late 1980s and early 1990s. The aim of the proposed schemas was to show that it is consistent with General Relativity that there is a physical process that computes something that is not computable on Turing machines. I will outline the basic idea using the schema proposed by Németi [206] that uses a highly probable assumption that there are large rotating black holes. Note that all schemas assume that the universe is infinite in size and time, or at least that the size of the universe grows beyond any limit.

   Consider two entities P (programmer) and C (computer), such that P wants to learn, using C, something that is not classically computable. Namely, P wants to know if some efficiently computable predicate $R(n)$ is satisfied by all numbers. For example, $R(n)$ can express that the nth nontrivial zero of the ζ-function is on the line $Re(x) = \frac{1}{2}$. Then $R(x)$ is satisfied by all numbers if and only if the Riemann Hypothesis is true. To this end we need a black hole (with suitable parameters) and let P fall into it (on a suitable trajectory). Meanwhile C will use a Turing machine to check all numbers for the property R. One cannot construct an infinite tape for the Turing machine, but since the universe is infinite, it is possible to gradually extend the tape beyond any bounds, which is all that C needs.

   From the point of view of C, P will never reach the event horizon of the black hole. So if C finds a number that does not satisfy R, it can send a signal to P while P is still before the event horizon.[27]

   From the point of view of P, nothing special happens at the event horizon. In particular, he will cross the event horizon in finite time according to his watch. But he can calculate time t_1 when he will be crossing the event horizon. So at time t_1 he will know that C tested all numbers (and ceased to exist). If everything is set up properly, at some time $t_2 > t_1$ P will know the answer: if a signal has arrived, then there is a number that violates R, otherwise R holds true for all numbers.

   This shows that such computations are consistent with general relativity. Though some effort has been made to include quantum theory into consideration, it is not clear that they are consistent with other theories that describe basic physical phenomena. Note that we still do not have a consistent theory that would unite relativity and quantum theory, so the concept of consistency with known physical phenomena is not precisely defined.

6. *Second-order logic is not axiomatizable.* This means that there is no formal system in which a second-order sentence would be provable if and only if it is sat-

[27] More precisely, we should talk about the event horizon as seen by C because C is in a finite distance from the black hole.

isfied by every second-order structure. This implies that higher order logics also cannot be axiomatized.

One can prove this fact by considering the complexity of the set of logically valid second-order sentences. Given a formal system Π, the set of sentences provable Π is recursively enumerable. Recall that recursively enumerable is the same as being Σ_1 in the arithmetical hierarchy. The set of sentences ϕ provable in Π can be defined by the condition informally stated as follows:

There exists a Π-proof of ϕ.

So there is only one existential quantifier, which explains Σ_1. But the set of second-order logically valid sentences is much more complex—it is not contained in any of the classes of the arithmetical hierarchy. Therefore it cannot be axiomatized.

The reason this set has such a large complexity is that in second-order logic one can define the standard model of arithmetic. Consequently, all true arithmetical sentences can be derived in second-order logic. In more detail, let Φ be the conjunction of the three second-order axioms of Dedekind-Peano Arithmetic (page 30) and the first seven axioms of Peano Arithmetic (page 116). Then for every first-order arithmetical sentence ψ, ψ is true if and only if the second-order sentence $\Phi \to \psi$ is logically valid. Thus for any arithmetical set X, the decision problem for X can be reduced to the decision problem of the set of logically valid second-order sentences.

2.5 The Lambda Calculus

The λ-calculus is a very interesting formal system whose main features are universality and flexibility. As such, it can be used for various purposes. It also has many versions which split into two main branches: the *type-free λ-calculus* and the *typed λ-calculus*. The λ-calculus can be used as a logical calculus, a formalism for defining algorithms and as a formal system for the foundations of mathematics. In the type-free λ-calculus all of the elements are of the same type. The typed λ-calculus is based on a hierarchy of functionals (see page 17), in a similar fashion as types are used in Russell's Theory of Types, which we will consider shortly, and thus it can be viewed as a generalization of Russell's theory. Because of important connections with proof theory, it is often viewed as a branch of it. The typed λ-calculus also has applications in computer science, especially in functional programming and automated theorem proving.

Church introduced the λ-calculus in the early 1930s [42]. A little later, he introduced the typed version [44]. Related calculi, called *combinatory logic* were invented independently by M.I. Schönfinkel and H.B. Curry a little before, but the connection between the two approaches was discovered later. As the title of his seminal paper '*A set of postulates for the foundation of logic*' suggests, Church intended to use his formal system for the foundations of mathematics in a similar way as Frege, Whitehead, Russell and other logicists did. Thus the λ-calculus was conceived as a kind of general logic. His first attempts were plagued with inconsistencies, but soon after that he found the sound version and proved its consistency.

We will start with the type-free version. The basic idea of the λ-calculus is that we can think of everything as being a function, an argument and a value at the same time. Computer programs and computer data are a good example. Since both are just some files, you can run a program on data, but you can also run a program on a file that is a program as well, which is sometimes very useful. You can take a file which is not a program and ask your computer to execute a file like a program, in which case the computer also does something, though it is hardly of any use. The paradigm that objects are functions, arguments and values at the same time, is similar to the one used in set theory, where an object is a set and an element at the same time. In a sense, it is the opposite of the structural approach to mathematics that we considered in Chap. 1 (while, in contrast to it, the typed λ-calculus is fully in line with it).

The λ-calculus has one binary operation as the only nonlogical symbol, the interpretation of which is application of a function to an argument. Usually no symbol is used for the binary operation of application, instead we simply use juxtaposition of the arguments. So ab means that a function a is applied to an argument b (outside of the λ-calculus the most common notation for this operation is $a(b)$, but juxtaposition is also frequently used). This notation stresses the identification of objects and functions. We think of functions in the λ-calculus as being unary, as we apply them only to one argument, but they can also be treated as functions of arbitrarily many arguments. For instance, if you want to apply f to x and y, then first apply f to x. Then, since the result is again a function, you can apply it to y. Formally, this is written as $(fx)y$, or simply fxy, using the convention that parenthesis grouped to the left are omitted.

The essence of the λ-calculus is the principle that *every term defines a function that is an object of the theory.* This is not such an obvious principle as it may look at the first glance and, in fact, in some context it may even be contradictory (recall the related Comprehension Principle). To formalize this principle, Church introduced an operator, denoted by λ, from which the name of the calculus stems. This operator can be applied to any variable x and any term t, not necessarily containing a variable x, resulting in the expression

$$\lambda x.t.$$

In this expression, λ binds the variable x, like a quantifier. The dot between x and t is a convenient way of separating the operator from the term. (Dots were used instead of parenthesis very often at the beginning of the 20th century.) The meaning of the expression is:

the function which to x assigns the value obtained by evaluating the term on x.

Example $\lambda x.x$ is the identity function; $\lambda x.y$ is the function constantly equal to y. Notice the difference between the expressions x and $\lambda x.x$—the first one is a variable and can represent any object, the second one is a constant symbol and it represents a unique object, the identity function.

The λ-notation is very useful in general, but, unfortunately, most mathematicians do not know about it, although in mathematical practice we often need to distinguish a function from its value.

Example One can distinguish a *number* which is a square x^2, from the quadratic *function* $\lambda x.x^2$. In the expression $\int_0^1 x^2\,dx$ we use dx to say that we integrate the quadratic function, but we do not use $x^2\,dx$ for this purpose elsewhere. If the λ-notation was used, we would have a more systematic notation. The integral above would be expressed by $\int_0^1 \lambda x.x^2$.

Combinatory Algebras

For main-stream mathematicians, it may be easier to grasp the main ideas of the λ-calculus using a class of first-order structures that are models of the λ-calculus. The *Extensional Combinatory Algebras* is a class of structures defined by the following axioms:

The Axiom of the Existence of at Least Two Elements *There are two different elements.*

This axiom in combination with the others ensures that there are infinitely many elements.

The Axiom of Extensionality *If* $fx = gx$ *for every* x, *then* $f = g$.

In words, if function f gives the same value as function g for every argument x, then $f = g$. This is essentially the same as the Axiom of Extensionality for sets.

The Axiom Schema of Combinatory Completeness *For every term* t *and every sequence of variables* $x_1, \dots, x_n$ *that includes all variables of* t, *the following is an axiom*: *There exists an* f *such that for every* $x_1, \dots, x_n$, $fx_1 \dots x_n = t$.

In words, every function that can be defined by a term is present in the universe. This schema formalizes the basic principle of the λ-calculus. Using the λ-notation it can be stated as the following schema.

***β*-conversion** *For every term* t,

$$(\lambda x_1 \dots \lambda x_n.t)x_1 \dots x_n = t.$$

In first-order logic we can view the expression $\lambda x_1 \dots \lambda x_n.t$ as a new constant introduced to represent the function f from the schema above.

Curry discovered that it suffices to take only three instances of the Axiom Schema of Combinatory Completeness. For these special cases, he introduced constant symbols denoting the particular functions. Then all other functions resulting from the schema can be written as terms using these basic functions, called *combinators*. The three combinators are denoted by **I**, **K**, **S** and the three instances of the Axiom Schema of Combinatory Completeness are:[28]

1. $\mathbf{I}x = x$
2. $(\mathbf{K}x)y = x$
3. $((\mathbf{S}x)y)z = (xz)(yz)$

The simplicity of this axiom system is, indeed, striking. Moreover, the first axiom of the three axioms above is redundant, as one can express **I** by **SKK** (an easy exercise).

In order to get some feeling for this calculus, it is worthwhile to interpret at least the two simplest combinators **I** and **K**. **I** is clearly the identity function. **K** is the function which on argument x produces the function that is constantly equal to x, (namely, if we apply $\mathbf{K}x$ to any y we get x). In the λ-notation the combinators **I**, **K** and **S** are defined by

$$\mathbf{I} = \lambda x.x, \qquad \mathbf{K} = \lambda x \lambda y.x, \qquad \mathbf{S} = \lambda x \lambda y \lambda z.(xz)(yz).$$

Nevertheless, to construct a model of an extensional combinatory algebra is not easy. The consistency of the λ-calculus was proved using combinatory means. The proof is based on a concept of a normal form of a term, which plays a central role in the theory. Since we can interpret the theory of extensional combinatory algebras in the λ-calculus, this implies, by the completeness theorem, that there is a model of extensional combinatory algebras. However, an explicit construction of such a model was found only much later by Dana Scott (see Notes).

The following is an important theorem in the type-free λ-calculus.

The Fixed Point Theorem *For every f, there exists g such that*

$$fg = g.$$

Proof Let $h = \lambda x.f(xx)$, let $g = hh$. Then

$$fg = f(hh) = \big(\lambda x.f(xx)\big)h = hh = g. \qquad \square$$

This looks like a meaningless manipulation with terms, but it is not. I will not explain this proof because its essence is self-reference which can be more easily explained in the context of logic, which I will do in Chap. 4.

[28] For the sake of clarity I do not use the convention of omitting parentheses here.

The λ-Calculus as Logic

To explain how the λ-calculus can be used as logic, it is better to use the typed λ-calculus. The objects of this calculus are also functions, but in contrast to the type-free version, we can apply a function to an argument only if the types match. Starting form some basic types, new types are formed by taking the set of all functions mapping objects of a type τ to objects of a type σ. This type is denoted by $\tau \rightarrow \sigma$. If a is of type $\tau \rightarrow \sigma$ and b is of type τ, we can apply a to b to obtain an element ab of type σ. In order to formalize functions of two (and more) variables, one can either introduce the product of types $\tau_1 \times \tau_2$ and use $\tau_1 \times \tau_2 \rightarrow \sigma$, or use the type $\tau_1 \rightarrow (\tau_2 \rightarrow \sigma)$.

In the standard formalization of first-order logic we distinguish formulas from terms. The reason is that formulas express the truth or the falsehood, while terms denote objects. If t is a closed numerical term, then it has a value in the domain of the numbers; if $t(x)$ is a numerical term with one free variable x, then $t(x)$ denotes a function of one variable, etc. If ϕ is a sentence, then the value of ϕ is either truth or falsehood; if $\phi(x)$ is a formula with one free variable x, then it represents a *propositional function* of one variable or a *set*, etc. Church observed that it is only the range of values that distinguishes these two concepts, and since one can consider various ranges, there is no need to distinguish the Boolean one from others.

So let us assume that we have the Boolean type $B = \{\mathbf{0}, \mathbf{1}\}$ and let us use only terms. Since we have eliminated sentences, we have to say what we will use instead of them and what will replace proofs. Sentences are easy: these are just the terms that are of type B. In order to define proofs, we have to view a proof not as a text that presents evidence that a sentence is true, but as a *process* by which we obtain this evidence. Consider a proof by contradiction of a sentence ϕ. It starts with writing $\neg\phi$, then we derive new formulas using logical rules and eventually we derive contradiction. This process is very much like evaluating a numerical expression: we start with the expression, apply rules to transform it, and eventually obtain the value. Again, there is no reason to treat these two kinds of process as being distinct. So in the λ-calculus we prove a sentence represented by a term t by evaluating it, which means that we apply rewriting rules until we obtain the Boolean value $\mathbf{1}$.

In order to embed first-order logic into the λ-calculus, we must define translations of connectives and quantifiers. Since connectives are, by definition, Boolean functions, their translations are straightforward. For example, negation is represented by a constant of type $B \rightarrow B$. Since quantification always ranges only over objects of a given type, we need a constant Π_τ for every type τ. This constant is of type $(\tau \rightarrow B) \rightarrow B$. Recall that $\tau \rightarrow B$ are propositional functions of one variable, functions defined by a formula $\phi(x)$ with one free variable x. Hence Π_τ maps such functions to the Boolean constants. The sentence $\forall x\ \phi(x)$ is true if and only if the propositional function is constantly equal to $\mathbf{1}$. So Π_τ is the function that assigns $\mathbf{1}$ to all functions constantly equal to $\mathbf{1}$, and $\mathbf{0}$ to all other functions.

Let a be a term representing a formula $\phi(x)$, where x is of type τ. Then

$$\forall x\ \phi(x) \quad \text{is represented by } \Pi_\tau \lambda x.a.$$

The use of the letter Π is quite natural: in the standard notation we would express the same by $\Pi_{x\in\tau}a(x)$.

Describing axioms and rules would take us to far afield and it is not interesting anyway because we only need to translate the usual axioms into the new formalism. Let us just observe that axioms are rules that rewrite an expression directly to **1**.

Formulas as Types

In the typed λ-calculus we need different combinators for different types. Thus we have one combinator $\mathbf{I}_\tau$ for every type τ, one combinator $\mathbf{K}_{\tau,\sigma}$ for every pair of types τ, σ and one combinator $\mathbf{S}_{\tau,\sigma,\rho}$ for every triple of types τ, σ, ρ. Let us look the types of these combinators.[29]

$$\begin{aligned}\mathbf{I}_\tau &: \tau \to \tau\\ \mathbf{K}_{\tau,\sigma} &: \tau \to (\sigma \to \tau)\\ \mathbf{S}_{\tau,\sigma,\rho} &: \big(\tau \to (\sigma \to \rho)\big) \to \big((\tau \to \sigma) \to (\tau \to \rho)\big)\end{aligned}$$

The striking fact is that the types look like propositional tautologies; indeed, if we interpret the type variables as propositional variables they are tautologies. Moreover, these three tautologies are often used as axioms of propositional calculus. And this is still not all, look at this:

If $a : \tau$ and $b : \tau \to \sigma$, then $ba : \sigma$.

This rule corresponds to the propositional rule of *modus ponens.* Thus suddenly, an axiomatization of a fragment of propositional logic pops up. The fragment is the restriction of intuitionistic logic to formulas built of implications, the *implicational fragment of intuitionistic logic.*

The connection with logic is very close. Not only these three combinators have types that are intuitionistic tautologies, but *all terms* have such types. *Vice versa*, for every tautology ϕ of the implicational fragment of intuitionistic logic, there exits a term whose type represents ϕ. Hence we can view terms as proofs of the tautologies that are represented by their types. This correspondence is represented by the following diagram.

$$\begin{aligned}\textit{type} &\leftrightarrow \textit{formula}\\ \textit{term} &\leftrightarrow \textit{proof}\end{aligned}$$

Viewing it semantically, as a complex structure, we have types that are empty, which correspond to non-tautologies, and types that are *inhabited*, which correspond to tautologies. To prove a tautology means to show that the corresponding type is inhabited.

[29] The column is used to express membership in a type.

Example In Curry's formalism variables and terms are used without specifying types. Then there are terms that do not have a type, hence cannot be interpreted as proofs. Show, as an exercise, that one can assign types to the combinators in the following term so that it has the specified type:

$$\big(\mathbf{S}(\mathbf{KS})\big)\mathbf{K} \,:\, (\sigma \to \rho) \to \big((\tau \to \sigma) \to (\tau \to \rho)\big).$$

The corresponding tautology is a version of the transitivity of implication. The term $(\mathbf{S}(\mathbf{KS}))\mathbf{K}$ is a proof of this tautology.

To extend this correspondence to full intuitionistic propositional logic, one has to use more operations on types, products $\tau \times \sigma$ for conjunctions, sums $\tau \oplus \sigma$ for disjunctions, and one has to add the empty type $\perp$ for *false*. (The negation of ϕ is represented by $\phi \to \perp$.)

This correspondence is called the *Curry-Howard isomorphism*. It has been extended in many directions. Concerning logic, the correspondence has been extended to first-order intuitionistic logic, classical propositional logic and some other logics. Concerning proof theory, connections have been found between certain transformations of proofs in logic on the one hand, and transformations of terms in the λ-calculus on the other. In this way the computational nature of the λ-calculus has been translated to proof theory and one can view certain proof-theoretical operations as computations.

Notes

1. *Composition of functions in the λ-calculus.* The binary operation of the λ-calculus, the application of a function to an argument, should not be confused with the composition of functions. The latter is definable by means of the combinator **B** defined by $((\mathbf{B}x)y)z = x(yz)$ (or by the λ-term $\lambda x \lambda y \lambda z.x(yz)$). The composition of function y with function x is the function $(\mathbf{B}x)y$. One can show that **B** can be expressed as $(\mathbf{S}(\mathbf{KS}))\mathbf{K}$.
2. *Combinatory logic.* The classical presentation of the λ-calculus does not include the Axiom of Extensionality. The system defined by the axioms 1.–3. on page 149 with the axiom $\mathbf{K} \neq \mathbf{S}$ is Curry's *Combinatory Logic*. It is slightly weaker than the λ-calculus, but it can be made equivalent to it by adding five more equalities. By adding four other equalities it can be made equivalent to the λ-calculus with the Axiom of Extensionality. Furthermore, there exists a presentation of the Combinatory Logic with a single combinator.
3. *The Church-Rosser Theorem.* The great appeal of the λ-calculus stems from the possibility of manipulations with terms, which can be used to define computations. In fact, the λ calculus was one of the first systems by means of which computable functions were defined. The two basic operations are *λ-abstraction* and *β-conversion*. The first of the two is the operation considered above: given a term we construct another term that represents the function defined by the given

term. β-conversion is in a sense the opposite operation; it is the transformation of a term $\lambda x.\tau(x)a$ into the term $\tau(a)$. This follows from the definition: applying the function defined by a term $\tau(x)$ to an argument a we must obtain the value of the term on the argument a. At first glance it looks stupid to introduce a λ-term in order to eliminate it in the next moment, but it is not as simple as it may appear, and quite interesting things may happen. The point is that τ may also contain occurrences of the λ-operator. Thus if we have a complex term, there may be several different places where we can apply β-conversion, hence we can reduce it in different ways. Furthermore, β-conversion may reduce the size of the term, but it also may increase it. (If σ is a long term and x occurs in τ more than once, the term $\tau(\sigma)$ is clearly longer than $\lambda x.\tau(x)\sigma$.) Consequently, the process of successive applying β-conversion may run indefinitely when started on some terms. The key result of the theory, the *Church-Rosser Theorem*, says that if the process converges, then it converges to a unique term, whatever strategy we choose. By converging we mean that we arrive at a term to which β-conversion is not applicable anymore. This uniquely determined term is called *the normal form*.

There are several interesting applications of this basic result. In particular, one can use it to prove the *consistency of the λ-calculus*. One can prove that two terms that have normal form are equal in the λ-calculus if and only if the normal forms are equal. Since it is easy to construct two different normal forms we get immediately the conclusion that the λ-calculus is consistent (meaning that one cannot prove that all elements are equal).

4. *The λ-calculus and computations.* In order to define computations, terms of the λ-calculus will be interpreted in two ways: as programs and as data. The computation of program τ on input data σ is the β-conversion process on the term $\tau\sigma$. The output is the resulting normal form, provided it exists, otherwise the computation diverges. To compute functions of more variables we apply program τ to a string of data $\sigma_1, \dots, \sigma_n$ as follows: $(\dots((\tau\sigma_1)\sigma_2)\dots\sigma_n)$ (according to the standard convention, the parentheses can be omitted).

   In order to compare it with other computational models, we need to encode some standard structures by terms that are in the normal form. One can define natural numbers $0, 1, 2, 3, \dots$ by the terms

$$\lambda x\lambda y.y, \quad \lambda x\lambda y.xy, \quad \lambda x\lambda y.x(xy), \quad \lambda x\lambda y.x\big(x(xy)\big), \quad \dots.$$

   So, for example, 0 is the function that on every argument gives the identity function as the value. It is a remarkable fact that every partial recursive function can be defined by a term. For example, addition can be defined by the term $\lambda x\lambda y\lambda z\lambda u.((xz)(yz))u$ and multiplication by $\lambda x\lambda y\lambda z.x(yz)$ (which is the combinator **B**).

5. *Propositional calculi from the λ-calculus.* It is interesting to see what the logical calculi resulting from the Curry-Howard isomorphism are. If we only use constant terms, as in the example of transitivity of implication, the calculus is essentially a Hilbert-style calculus (also called Frege system). If we allow variables and λ-abstraction, the proof system is like the natural deduction calculus.

A variable plays the role of an assumption and when λ-abstraction is applied to it, the assumption is discharged.

β-reduction has a different role. It corresponds to normalization of proofs, a transformation related to cut-elimination (which we will consider in Chap. 6).

6. *A model of the λ-calculus.* The first natural model of the λ-calculus was constructed by D. Scott in 1971 [261]. Scott found a general construction that can be applied to any nontrivial complete lattice. We will only consider the simplest case, where one starts with the two element lattice. We define a sequence of lattices $D_0, D_1, D_2, \dots$ as follows. D_0 is the two element lattice. Having D_n, define D_{n+1} as the set of all order preserving mappings $f : D_n \to D_n$ with the ordering given by the condition that $f \leq g$ if $f(x) \leq g(x)$ for all $x \in D_n$. Furthermore, define order-preserving mappings $\phi_n : D_n \to D_{n+1}$ and $\psi_n : D_{n+1} \to D_n$ as follows. $\phi_0(x)$ is the constant function equal to x, $\psi_0(x)$ is $x(\bot)$, where $\bot$ is the bottom element of D_0. Having ϕ_n and ψ_n, define ϕ_{n+1} and ψ_{n+1} as follows. For $x \in D_{n+1}$, i.e., $x : D_n \to D_n$, put

$$\phi_{n+1}(x) = \psi_n \circ x \circ \psi_n,$$

and for $y \in D_{n+2}$, (i.e., $y : D_{n+1} \to D_{n+1}$) put

$$\psi_{n+1}(y) = \psi_n \circ y \circ \phi_n,$$

where $\circ$ denotes composition of functions. Note that ϕ_n is an embedding, ψ_n is a mapping onto D_n, and $\psi_n \circ \phi_n$ is the identity. Put

$$D_\infty = \big\{(x_0, x_1, \dots);\ \forall n\ \psi_n(x_{n+1}) = x_n\big\}.$$

D_∞ is the universe of the algebra. It is a limit of D_n's in a well defined sense. In particular, every D_n is naturally embedded in D_∞ by the assignment

$$\iota_n(x) = \big(\psi_n^n(x), \dots, \psi_n^2(x), \psi_n(x), x, \phi_n(x), \phi_n^2(x), \dots\big).$$

Furthermore, D_∞ with the natural ordering is a complete lattice. We denote the supremum of a set $X \subseteq D_\infty$ by $\bigvee X$; to compute it, take suprema on all coordinates. Now we are ready to define the operation of application. For $a, b \in D_\infty$, where $a = (a_0, a_1, \dots), b = (b_0, b_1, \dots)$,

$$ab = \bigvee_n \iota_n\big(a_{n+1}(b_n)\big).$$

The trick is that we can think of an element of D_∞ as a sequence of elements and as a sequence of functions at the same time. (It is tempting to define the nth coordinate of ab simply as $a_{n+1}(b_n)$, but this does not work.)

To prove the combinatory completeness, one needs to have a natural property that is satisfied by all functions that are definable by terms of the algebra. To this end Scott uses a natural topology defined on D_∞, and the corresponding concept of continuous functions. Continuous functions are characterized as the functions that preserve suprema of directed sets. (A set is directed if for every two elements there is an element which is larger than both.) For finite lattices, continuous functions are the order preserving functions. So in the simplified case here, where all D_n's are finite, we only needed the property of order preserving.

It is possible to prove that

(i) The operation $a, b \mapsto ab$ is continuous in D_∞.
(ii) For every continuous function $f : D_\infty \mapsto D_\infty$, there exists $a \in D_\infty$ such that for all $x \in D_\infty$, $f(x) = ax$.

In order to get the combinatory completeness, we need a little more in (ii). The function f may depend on other parameters, and then a is a function of these parameters. We need a, as a function depending on these parameters, to be continuous. Then the two statements imply combinatory completeness.

Main Points of the Chapter

- The language of mathematics developed by converging to a minimal and universal language.
- The primitives of a logical language are: constants, variables, relation symbols, connectives and quantifiers. The syntax uses the same principles as the syntax of natural languages.
- The truth of a sentence in a structure can be precisely defined.
- Consequently, the logical validity of a sentence can be defined by the condition that the sentence is true in all structures.
- The concept of a proof can be made quite precise by postulating syntactical rules that must be satisfied by every proof.
- The logical calculus is complete: every sentence that is true in all structures can be proved.
- People are willing to use a completely formal language, if there is a strong incentive to do so.
- We have several precise mathematical concepts that define a class of computable functions; for example, Turing machines. We believe that this class comprises all computable functions (the Church-Turing Thesis), but we cannot prove it, as it is not a mathematical statement.
- There are very concrete noncomputable functions. The problem is not that our devices are too inefficient or that we do not have enough time to compute these functions, it is because they are not computable even in principle.
- There are universal Turing machines that can simulate every Turing machine. Computers are constructed so that they are universal in this sense.

Springer

http://www.springer.com/978-3-319-00118-0

Logical Foundations of Mathematics and Computational
Complexity
A Gentle Introduction
Pudlák, P.
2013, XIV, 695 p. 49 illus., 4 illus. in color., Hardcover
ISBN: 978-3-319-00118-0