Chapter 6

Logic

6.1 Introduction

In Chapter 2 we have introduced some basic concepts of logic, but the treatment was quite informal. In this chapter we discuss the foundations of logic in a mathematically rigorous manner. In particular, we clearly distinguish between the syntax and the semantics of a logic and between syntactic derivations of formulas and logical consequences they imply. We also introduce the concept of a logical calculus and define soundness and completeness of a calculus. Moreover, we discuss in detail a concrete calculus for propositional logic, the so-called resolution calculus.

At a very general level, the goal of logic is to provide a framework for expressing mathematical statements and for expressing and verifying proofs for such statements. A more ambitious, secondary goal can be to provide tools for *automatically* or semi-automatically generating a proof for a given statement.

A treatment of logic usually begins with a chapter on propositional logic[1] (see Section 6.5), followed by a chapter on predicate (or first-order) logic[2] (see Section 6.6), which can be seen as an extension of propositional logic. There are several other logics which are useful in Computer Science and in mathematics, including temporal logic, modal logic, intuitionistic logic, and logics for reasoning about knowledge and about uncertainty. Most if not all relevant logics contain the logical operators from propositional logic, i.e., $\wedge, \vee, \neg$ (and the derived operators $\rightarrow$ and $\leftrightarrow$), as well as the quantifiers ($\forall$ and $\exists$) from predicate logic.

Our goal is to present the general concepts that apply to all types of logics in a unified manner, and then to discuss the specific instantiations of these

[1]German: Aussagenlogik
[2]German: Prädikatenlogik

concepts for each logic individually. Therefore we begin with such a general treatment (see Sections 6.2, 6.3, and 6.4) before discussing propositional and predicate logic. From a didactic viewpoint, however, it will be useful to switch back and forth between the generic concepts of Sections 6.2, 6.3, and 6.4 and the concrete instantiations of Sections 6.5 and 6.6.

We give a general warning: Different treatments of logic often use slightly or sometimes substantially different notation.[3] Even at the conceptual level there are significant differences. One needs to be prepared to adopt a particular notation used in a particular application context. However, the general principles explained here are essentially standard.

We also refer to the book by Kreuzer and Kühling and that by Schöning mentioned in the preface of these lecture notes.

6.2 Proof Systems

6.2.1 Definition

In a formal treatment of mathematics, all objects of study must be described in a well-defined syntax. Typically, syntactic objects are finite strings over some alphabet Σ, for example the symbols allowed by the syntax of a logic or simply the alphabet $\{0,1\}$, in which case syntactic objects are bit-strings. Recall that Σ^* denotes the set of finite strings of symbols from Σ.

In this section, the two types of mathematical objects we study are *mathematical statements* of a certain type and *proofs* for this type of statements. By a statement type we mean for example the class of statements of the form that a given number n is prime, or the class of statements of the form that a given graph G has a Hamiltonian cycle (see below), or the class of statements of the form that a given formula F in propositional logic is satisfiable.

Consider a fixed type of statements. Let $\mathcal{S} \subseteq \Sigma^*$ be the set of (syntactic representations of) mathematical statements of this type, and let $\mathcal{P} \subseteq \Sigma^*$ be the set of (syntactic representations of) proof strings.[4]

Every statement $s \in \mathcal{S}$ is either true or false. The *truth function*

$$\tau : \mathcal{S} \rightarrow \{0,1\}$$

assigns to each $s \in \mathcal{S}$ its truth value $\tau(s)$. This function τ defines the meaning, called the *semantics*, of objects in $\mathcal{S}$.[5]

[3]For example, in some treatments the symbol $\Rightarrow$ is used for $\rightarrow$, which can be confusing.
[4]Membership in $\mathcal{S}$ and also in $\mathcal{P}$ is assumed to be efficiently checkable (for some notion of efficiency).
[5]In the context of logic discussed from the next section onwards, the term semantics is used in a specific restricted manner that is compatible with its use here.

An element $p \in \mathcal{P}$ is either a (valid) proof for a statement $s \in \mathcal{S}$, or it is not. This can be defined via a *verification function*

$$\phi : \mathcal{S} \times \mathcal{P} \rightarrow \{0,1\},$$

where $\phi(s,p) = 1$ means that p is a valid proof for statement s.

Without strong loss of generality we can in this section consider

$$\mathcal{S} = \mathcal{P} = \{0,1\}^*,$$

with the understanding that any string in $\{0,1\}^*$ can be interpreted as a statement by defining syntactically wrong statements as being false statements.

Definition 6.1. A *proof system*[6] is a quadruple $\Pi = (\mathcal{S}, \mathcal{P}, \tau, \phi)$, as above.

We now discuss the two fundamental requirements for proof systems.

Definition 6.2. A proof system $\Pi = (\mathcal{S}, \mathcal{P}, \tau, \phi)$ is *sound*[7] if no false statement has a proof, i.e., if for all $s \in \mathcal{S}$ for which there exists $p \in \mathcal{P}$ with $\phi(s,p) = 1$, we have $\tau(s) = 1$.

Definition 6.3. A proof system $\Pi = (\mathcal{S}, \mathcal{P}, \tau, \phi)$ is *complete*[8] if every true statement has a proof, i.e., if for all $s \in \mathcal{S}$ with $\tau(s) = 1$, there exists $p \in \mathcal{P}$ with $\phi(s,p) = 1$.

In addition to soundness and completeness, one requires that the function ϕ be *efficiently computable* (for some notion of efficiency).[9] We will not make this formal, but it is obvious that a proof system is useless if proof verification is computationally infeasible. Since the verification has to generally examine the entire proof, the length of the proof cannot be infeasibly long.[10]

6.2.2 Examples

Example 6.1. An undirected *graph* consists of a set V of nodes and a set E of edges between nodes. Suppose that $V = \{0, \ldots, n-1\}$. A graph can then be described by the so-called *adjacency matrix*, an $n \times n$-matrix M with $\{0,1\}$-entries,

[6]The term proof system is also used in different ways in the mathematical literature.

[7]German: korrekt

[8]German: vollständig

[9]The usual efficiency notion in Computer Science is so-called *polynomial-time computable* which we do not discuss further.

[10]An interesting notion introduced in 1998 by Arora et al. is that of a *probabilistically checkable proof (PCP)*. The idea is that the proof can be very long (i.e., exponentially long), but that the verification only examines a very small random selection of the bits of the proof and nevertheless can decide correctness, except with very small error probability.

where $M_{i,j} = 1$ if and only if there is an edge between nodes i and j. A graph with n nodes can hence be represented by a bit-string of length n^2, by reading out the entries of the matrix row by row.

We are now interested in proving that a given graph has a so-called *Hamiltonian cycle*, i.e., that there is a closed path from node 1 back to node 1, following edges between nodes, and visiting every node exactly once. We are also interested in the problem of proving the negation of this statement, i.e., that a given graph has *no* Hamiltonian cycle. Deciding whether or not a given graph has a Hamiltonian cycle is considered a computationally very hard decision problem (for large graphs).[11]

To prove that a graph has a Hamiltonian cycle, one can simply provide the sequence of nodes visited by the cycle. A value in $V = \{0, \ldots, n-1\}$ can be represented by a bit-string of length $\lceil \log_2 n \rceil$, and a sequence of n such numbers can hence be represented by a bit-string of length $n\lceil \log_2 n \rceil$. We can hence define $\mathcal{S} = \mathcal{P} = \{0,1\}^*$.

Now we can let τ be the function defined by $\tau(s) = 1$ if and only if $|s| = n^2$ for some n and the n^2 bits of s encode the adjacency matrix of a graph containing a Hamiltonian cycle. If $|s|$ is not a square or if s encodes a graph without a Hamiltonian cycle, then $\tau(s) = 0$.[12] Moreover, we can let ϕ be the function defined by $\phi(s,p) = 1$ if and only if, when s is interpreted as an $n \times n$-matrix M and when p is interpreted as a sequence of n different numbers $(a_1, \ldots, a_n)$ with $a_i \in \{0, \ldots, n-1\}$ (each encoded by a bit-string of length $\lceil \log_2 n \rceil$), then the following is true:

$$M_{a_i, a_{i+1}} = 1$$

for $i = 1, \ldots, n-1$ and

$$M_{a_n, a_1} = 1.$$

This function ϕ is efficiently computable. The proof system is sound because a graph without Hamiltonian cycle has no proof, and it is complete because every graph with a Hamiltonian cycle has a proof. Note that each s with $\tau(s) = 1$ has at least n different proofs because the starting point in the cycle is arbitrary.

Example 6.2. Let us now consider the opposite problem of proving the inexistence of a Hamiltonian cycle in a given graph. In other words, in the above example we define $\tau(s) = 1$ if and only if $|s| = n^2$ for some n and the n^2 bits of s encode the adjacency matrix of a graph *not* containing Hamiltonian cycle. In this case, no sound and complete proof system (with reasonably short and efficiently verifiable proofs) is known. It is believed that no such proof system exists.

[11]The best known algorithm has running time exponential in n. The problem is actually NP-complete, a concept that will be discussed in a later course on theoretical Computer Science.

[12]Note that τ defines the meaning of the strings in $\mathcal{S}$, namely that they are meant to encode graphs and that we are interested in whether a given graph has a Hamiltonian cycle.

Example 6.3. Let again $\mathcal{S} = \mathcal{P} = \{0,1\}^*$, and for $s \in \{0,1\}^*$ let $n(s)$ denote the natural number whose (standard) binary representation is s, with the convention that leading 0's are ignored. (For example, $n(101011) = 43$ and $n(00101) = 5$.) Now, let τ be the function defined as follows: $\tau(s) = 1$ if and only if $n(s)$ is *not* a prime number. Moreover, let ϕ be the function defined by $\phi(s,p) = 1$ if and only if $n(s) = 0$, or if $n(s) = 1$, or if $n(p)$ divides $n(s)$ and $1 < n(p) < n(s)$. This function ϕ is efficiently computable. This is a proof system for the non-primality (i.e., compositeness) of natural numbers. It is sound because every s corresponding to a prime number $n(s)$ has no proof since $n(s) \neq 0$ and $n(s) \neq 1$ and $n(s)$ has no divisor d satisfying $1 < d < n(s)$. The proof system is complete because every natural number n greater than 1 is either prime or has a prime factor q satisfying $1 < q < n$ (whose binary representation can serve as a proof).

Example 6.4. Let us consider the opposite problem, i.e., proving primality of a number $n(s)$ represented by s. In other words, in the previous example we replace "not a prime" by "a prime". It is far from clear how one can define a verification function ϕ such that the proof system is sound and complete. However, such an efficiently computable function ϕ indeed exists. Very briefly, the proof that a number $n(s)$ (henceforth we simply write n) is prime consists of (adequate representations of):

1) the list $p_1, \ldots, p_k$ of distinct prime factors of $n-1$,
2) a (recursive) proof of primality for each of $p_1, \ldots, p_k$[13]
3) a generator g of the group $\mathbb{Z}_n^*$.

The exact representation of these three parts of the proof would have to be made precise, but we omit this here as it is obvious how this could be done.

The verification of a proof (i.e., the computation of the function ϕ) works as follows.

- If $n = 2$ or $n = 3$, then the verification stops and returns 1.[14]
- It is tested whether $p_1, \ldots, p_k$ all divide $n-1$ and whether $n-1$ can be written as a product of powers of $p_1, \ldots, p_k$ (i.e., whether $n-1$ contains no other prime factor).
- It is verified that
$$g^{n-1} \equiv_n 1$$
and, for all $i \in \{1, \ldots, k\}$, that
$$g^{(n-1)/p_i} \not\equiv_n 1.$$

[13]recursive means that the same principle is applied to prove the primality of every p_i, and again for every prime factor of $p_i - 1$, etc.

[14]One could also consider a longer list of small primes for which no recursive primality proof is required.

(This means that g has order $n-1$ in $\mathbb{Z}_n^*$).

- For every p_i, an analogous proof of its primality is verified (recursively).

This proof system for primality is sound because if n is not a prime, then there is no element of $\mathbb{Z}_n^*$ of order $n-1$ since the order of any element is at most $\varphi(n)$, which is smaller than $n-1$. The proof system is complete because if n is prime, then $GF(n)$ is a finite field and the multiplicative group of any finite field, i.e., $\mathbb{Z}_n^*$, is cyclic and has a generator g. (We did not prove this statement in this course.)[15]

6.2.3 Discussion

The examples demonstrate the following important points:

- While proof verification must be efficient (in some sense not defined here), *proof generation* is generally not (or at least not known to be) efficient. For example, finding a proof for the Hamiltonian cycle example requires to find such a cycle, a problem that, as mentioned, is believed to be very hard. Similarly, finding a primality proof as discussed would require the factorization of $n-1$, and the factoring problem is believed to be hard. In general, finding a proof (if it exists) is a process requiring insight and ingenuity, and it cannot be efficiently automated.
- A proof system is always restricted to a certain type of mathematical statement. For example, the proof system of Example 6.1 is very limited in the sense that it only allows to prove statements of the form "graph G has a Hamiltonian cycle".
- Proof verification can in principle proceed in very different ways. The proof verification method of logic, based on checking a sequence of rule applications, is (only) a special case.
- Asymmetry of statements and their negation: Even if a proof system exists for a certain type of statements, it is quite possible that for the negation of the statements, no proof system (with efficient verification) exists.

6.2.4 Proof Systems in Theoretical Computer Science *

The concept of a proof system appears in a more concrete form in theoretical computer science (TCS), as follows. Statements and proofs are bit-strings, i.e., $\mathcal{S} = \mathcal{P} = \{0,1\}^*$. The predicate τ defines the set $L \subseteq \{0,1\}^*$ of strings that correspond to true statements:

[15]Actually, a quite efficient deterministic primality test was recently discovered by Agrawal et al., and this means that primality can be checked without a proof. In other words, there exists a trivial proof system for primality with empty proofs. However, this fact is mathematically considerably more involved than the arguments presented here for the soundness and completeness of the proof system for primality.

$$L = \{s \mid \tau(s) = 1\}.$$

Conversely, every subset $L \subseteq \{0,1\}^*$ defines a predicate τ. In TCS, such a set L of strings is called a *formal language*, and one considers the problem of proving that a given string s is in the language, i.e., $s \in L$. A proof for $s \in L$ is called a *witness* of s, often denoted as w, and the verification function $\phi(s, w)$ defines whether a string w is a witness for $s \in L$.

One then considers the special case where the length of w is bounded by a polynomial of the length of s and where the function ϕ must be computable in polynomial time, i.e., by a program with worst-case running time polynomial in the length of s. Then, the important class NP of languages is the set of languages for which such a polynomial-time computable verification function exists.

As mentioned in a footnote, a type of proof system of special interest are so-called *probabilistically checkable proofs (PCP)*.

An important extension of the concept of proof systems are so-called *interactive proofs*.[16] In such a system, the proof is not a bit-string, but it consists of an interaction (a protocol) between the prover and the verifier, where one tolerates an immensely small (e.g. exponentially small) probability that a verifier accepts a "proof" for a false statement. The reason for considering such interactive proofs are:

- Such interactive proofs can exist for statements for which a classical (non-interactive) proof does not exist. For example, there exists an interactive proof system for the *non*-Hamiltonicity of graphs.
- Such interactive proofs can have a special property, called *zero-knowledge*, which means that the verifier learns absolutely nothing (in a well-defined sense) during the protocol, except that the statement is true. In particular, the verifier cannot prove the statement to somebody else.
- Interactive proofs are of crucial importance in a large number of applications, especially if they have the zero-knowledge property, for example in sophisticated block-chain systems.

6.3 Elementary General Concepts in Logic

The purpose of this section is to introduce the most basic concepts in logic in a general manner, not specific to a particular logic. However, this section is best appreciated by considering concrete examples of logics, in particular propositional logic and predicate logic. Without discussing such examples in parallel to introducing the concepts, this section will be hard to appreciate. We will discuss the general concepts and the concrete examples in parallel, going back and forth between Section 6.3 and Sections 6.5 and 6.6.

[16]This topic is discussed in detail in the Master-level course *Cryptographic Protocols* taught by Martin Hirt and Ueli Maurer.

6.3.1 The General Goal of Logic

A goal of logic is to provide a specific proof system $\Pi = (\mathcal{S}, \mathcal{P}, \tau, \phi)$ for which a very large class of mathematical statements can be expressed as an element of $\mathcal{S}$.

However, such a proof system $\Pi = (\mathcal{S}, \mathcal{P}, \tau, \phi)$ can never capture *all* possible mathematical statements. For example, it usually does not allow to capture (self-referential) statements about Π, such as "Π is complete", as an element of $\mathcal{S}$. The use of common language is therefore unavoidable in mathematics and logic (see also Section 6.7).

In logic, an element $s \in \mathcal{S}$ consists of one or more formulas (e.g. a formula, or a set of formulas, or a set of formulas and a formula), and a proof consists of applying a certain sequence of syntactic steps, called a *derivation* or a *deduction*. Each step consists of applying one of a set of allowed syntactic rules, and the set of allowed rules is called a *calculus*. A rule generally has some place-holders that must be instantiated by concrete values.

In standard treatments of logic, the syntax of $\mathcal{S}$ and the semantics (the function τ) are carefully defined. In contrast, the function ϕ, which consists of verifying the correctness of each rule application step, is not completely explicitly defined. One only defines rules, but for example one generally does not define a syntax for expressing how the place-holders of the rules are instantiated.[17]

6.3.2 Syntax, Semantics, Interpretation, Model

A *logic* is defined by the *syntax* and the *semantics*. The basic concept in any logic is that of a *formula*[18].

Definition 6.4. The *syntax* of a logic defines an alphabet Λ (of allowed symbols) and specifies which strings in Λ^* are formulas (i.e., are syntactically correct).

The semantics (see below) defines under which "conditions" a formula is *true* (denoted as 1) or *false* (denoted as 0).[19] What we mean by "conditions" needs to be made more precise and requires a few definitions.

Some of the symbols in Λ (e.g. the symbols A and B in propositional logic or the symbols P and Q in predicate logic) are understood as variables, each of which can take on a value in a certain domain associated to the symbol.

[17]In a fully computerized system, this must of course be (and indeed is) defined.

[18]German: Formel

[19]There are logics (not considered here) with more than two truth values, for example a logic with confidence or belief values indicating the degree of confidence in the truth of a statement.