2 Theoretical Background

Alice: *Will we ever get to the real stuff?*
Vittorio: *Cine nu cunoaşte lema, nu cunoaşte teorema.*
Riccardo: *What is Vittorio talking about?*
Sergio: *This is an old Romanian saying that means, "He who doesn't know the* lemma *doesn't know the* teorema."
Alice: *I see.*

This chapter gives a brief review of the main theoretical tools and results that are used in this volume. It is assumed that the reader has a degree of maturity and familiarity with mathematics and theoretical computer science. The review begins with some basics from set theory, including graphs, trees, and lattices. Then, several topics from automata and complexity theory are discussed, including finite state automata, Turing machines, computability and complexity theories, and context-free languages. Finally basic mathematical logic is surveyed, and some remarks are made concerning the specializing assumptions typically made in database theory.

2.1 Some Basics

This section discusses notions concerning binary relations, partially ordered sets, graphs and trees, isomorphisms and automorphisms, permutations, and some elements of lattice theory.

A *binary relation* over a (finite or infinite) set S is a subset R of $S \times S$, the cross-product of S with itself. We sometimes write $R(x, y)$ or xRy to denote that $(x, y) \in R$.

For example, if Z is a set, then inclusion ($\subseteq$) is a binary relation over the power set $\mathcal{P}(Z)$ of Z and also over the *finitary power set* $\mathcal{P}^{\text{fin}}(Z)$ of Z (i.e., the set of all finite subsets of Z). Viewed as sets, the binary relation $\leq$ on the set $\mathbf{N}$ of nonnegative integers properly contains the relation $<$ on $\mathbf{N}$.

We also have occasion to study n-ary relations over a set S; these are subsets of S^n, the cross-product of S with itself n times. Indeed, these provide one of the starting points of the relational model.

A binary relation R over S is *reflexive* if $(x, x) \in R$ for each $x \in S$; it is *symmetric* if $(x, y) \in R$ implies that $(y, x) \in R$ for each $x, y \in S$; and it is *transitive* if $(x, y) \in R$ and $(y, z) \in R$ implies that $(x, z) \in R$ for each $x, y, z \in S$. A binary relation that is reflexive, symmetric, and transitive is called an *equivalence relation*. In this case, we associate to each $x \in S$ the *equivalence class* $[x]_R = \{y \in S \mid (x, y) \in R\}$.

An example of an equivalence relation on **N** is *modulo* for some positive integer n, where $(i, j) \in \text{mod}_n$ if the absolute value $|i - j|$ of the difference of i and j is divisible by n.

A *partition* of a nonempty set S is a family of sets $\{S_i \mid i \in I\}$ such that (1) $\cup_{i \in I} S_i = S$, (2) $S_i \cap S_j = \emptyset$ for $i \neq j$, and (3) $S_i \neq \emptyset$ for $i \in I$. If R is an equivalence relation on S, then the family of equivalence classes over R is a partition of S.

Let E and E' be equivalence relations on a nonempty set S. E is a *refinement* of E' if $E \subseteq E'$. In this case, for each $x \in S$ we have $[x]_E \subseteq [x]_{E'}$, and, more precisely, each equivalence class of E' is a disjoint union of one or more equivalence classes of E.

A binary relation R over S is *irreflexive* if $(x, x) \notin R$ for each $x \in S$.

A binary relation R is *antisymmetric* if $(y, x) \notin R$ whenever $x \neq y$ and $(x, y) \in R$. A *partial order* of S is a binary relation R over S that is reflexive, antisymmetric, and transitive. In this case, we call the ordered pair (S, R) a *partially ordered set*. A *total order* is a partial order R over S such that for each $x, y \in S$, either $(x, y) \in R$ or $(y, x) \in R$.

For any set Z, the relation $\subseteq$ over $\mathcal{P}(Z)$ is a partially ordered set. If the cardinality $|Z|$ of Z is greater than 1, then this is not a total order. $\leq$ on **N** is a total order.

If (S, R) is a partially ordered set, then a *topological sort* of S (relative to R) is a binary relation R' on S that is a total order such that $R' \supseteq R$. Intuitively, R' is compatible with R in the sense that xRy implies $xR'y$.

Let R be a binary relation over S, and **P** be a set of properties of binary relations. The **P**-*closure* of R is the smallest binary relation R' such that $R' \supseteq R$ and R' satisfies all of the properties in **P** (if a unique binary relation having this specification exists). For example, it is common to form the transitive closure of a binary relation or the reflexive and transitive closure of a binary relation. In many cases, a closure can be constructed using a recursive procedure. For example, given binary relation R, the transitive closure R^+ of R can be obtained as follows:

1. If $(x, y) \in R$ then $(x, y) \in R^+$;
2. If $(x, y) \in R^+$ and $(y, z) \in R^+$ then $(x, z) \in R^+$; and
3. Nothing is in R^+ unless it follows from conditions (1) and (2).

For an arbitrary binary relation R, the reflexive, symmetric, and transitive closure of R exists and is an equivalence relation.

There is a close relationship between binary relations and graphs. The definitions and notation for graphs presented here have been targeted for their application in this book. A (*directed*) *graph* is a pair $G = (V, E)$, where V is a finite set of *vertexes* and $E \subseteq V \times V$. In some cases, we define a graph by presenting a set E of edges; in this case, it is understood that the vertex set is the set of endpoints of elements of E.

A *directed path* in G is a nonempty sequence $p = (v_0, \ldots, v_n)$ of vertexes such that $(v_i, v_{i+1}) \in E$ for each $i \in [0, n-1]$. This path is from v_0 *to* v_n and has length n. An *undirected path* in G is a nonempty sequence $p = (v_0, \ldots, v_n)$ of vertexes such that $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ for each $i \in [0, n-1]$. A (directed or undirected) path is *proper* if $v_i \neq v_j$ for each $i \neq j$. A (*directed* or *undirected*) *cycle* is a (directed or undirected, respectively) path $v_0, \ldots, v_n$ such that $v_n = v_0$ and $n > 0$. A directed cycle is proper if $v_0, \ldots, v_{n-1}$ is a proper path. An undirected cycle is proper if $v_0, \ldots, v_{n-1}$ is a proper path and $n > 2$. If G has a cycle from v, then G has a proper cycle from v. A graph $G = (V, E)$ is *acyclic* if it has no cycles or, equivalently, if the transitive closure of E is irreflexive.

Any binary relation over a finite set can be viewed as a graph. For any finite set Z, the graph $(\mathcal{P}(Z), \subseteq)$ is acyclic. An interesting directed graph is (M, L), where M is the set of metro stations in Paris and $(s_1, s_2) \in L$ if there is a train in the system that goes from s_1 to s_2 without stopping in between. Another directed graph is (M, L'), where $(s_1, s_2) \in L'$ if there is a train that goes from s_1 to s_2, possibly with intermediate stops.

Let $G = (V, E)$ be a graph. Two vertexes u, v are *connected* if there is an undirected path in G from u to v, and they are *strongly connected* if there are directed paths from u to v and from v to u. Connectedness and strong connectedness are equivalence relations on V. A (*strongly*) *connected component* of G is an equivalence class of V under (strong) connectedness. A graph is (strongly) connected if it has exactly one (strongly) connected component.

The graph (M, L) of Parisian metro stations and nonstop links between them is strongly connected. The graph $(\{a, b, c, d, e\}, \{(a, b), (b, a), (b, c), (c, d), (d, e), (e, c)\})$ is connected but not strongly connected.

The *distance* $d(a, b)$ of two nodes a, b in a graph is the length of the shortest path connecting a to b [$d(a, b) = \infty$ if a is not connected to b]. The *diameter* of a graph G is the maximum finite distance between two nodes in G.

A *tree* is a graph that has exactly one vertex with no in-edges, called the *root*, and no undirected cycles. For each vertex v of a tree there is a unique proper path from the root to v. A *leaf* of a tree is a vertex with no outedges. A tree is connected, but it is not strongly connected if it has more than one vertex. A *forest* is a graph that consists of a set of trees. Given a forest, removal of one edge increases the number of connected components by exactly one.

An example of a tree is the set of all descendants of a particular person, where (p, p') is an edge if p' is the child of p.

In general, we shall focus on directed graphs, but there will be occasions to use undirected graphs. An *undirected graph* is a pair $G = (V, E)$, where V is a finite set of vertexes and E is a set of two-element subsets of V, again called *edges*. The notions of path and connected generalize to undirected graphs in the natural fashion.

An example of an undirected graph is the set of all persons with an edge $\{p, p'\}$ if p is married to p'. As defined earlier, a tree $T = (V, E)$ is a directed graph. We sometimes view T as an undirected graph.

We shall have occasions to *label* the vertexes or edges of a (directed or undirected) graph. For example, a *labeling* of the vertexes of a graph $G = (V, E)$ with label set L is a function $\lambda : V \to L$.

Let $G = (V, E)$ and $G' = (V', E')$ be two directed graphs. A function $h : V \to V'$ is a *homomorphism* from G to G' if for each pair $u, v \in V$, $(u, v) \in E$ implies $(h(u), h(v)) \in E'$. The function h is an *isomorphism* from G to G' if h is a one-one onto mapping from V to V', h is a homomorphism from G to G', and h^{-1} is a homomorphism from G' to G. An *automorphism* on G is an isomorphism from G to G. Although we have defined these terms for directed graphs, they generalize in the natural fashion to other data and algebraic structures, such as relations, algebraic groups, etc.

Consider the graph $G = (\{a, b, c, d, e\}, \{(a, b), (b, a), (b, c), (b, d), (b, e), (c, d), (d, e), (e, c)\})$. There are three automorphisms on G: (1) the identity; (2) the function that maps c to d, d to e, e to c and leaves a, b fixed; and (3) the function that maps c to e, d to c, e to d and leaves a, b fixed.

Let S be a set. A *permutation* of S is a one-one onto function $\rho : S \rightarrow S$. Suppose that $x_1, \ldots, x_n$ is an arbitrary, fixed listing of the elements of S (without repeats). Then there is a natural one-one correspondence between permutations ρ on S and listings $x_{i_1}, \ldots, x_{i_n}$ of elements of S without repeats. A permutation ρ' is *derived* from permutation ρ by an *exchange* if the listings corresponding to ρ and ρ' agree everywhere except at some positions i and $i + 1$, where the values are exchanged. Given two permutations ρ and ρ', ρ' can be derived from ρ using a finite sequence of exchanges.

2.2 Languages, Computability, and Complexity

This area provides one of the foundations of theoretical computer science. A general reference for this area is [LP81]. References on automata theory and languages include, for instance, the chapters [BB91, Per91] of [Lee91] and the books [Gin66, Har78]. References on complexity include the chapter [Joh91] of [Lee91] and the books [GJ79, Pap94].

Let Σ be a finite set called an *alphabet*. A *word* over alphabet Σ is a finite sequence $a_1 \ldots a_n$, where $a_i \in \Sigma$, $1 \leq i \leq n$, $n \geq 0$. The *length* of $w = a_1 \ldots a_n$, denoted $|w|$, is n. The empty word ($n = 0$) is denoted by ϵ. The *concatenation* of two words $u = a_1 \ldots a_n$ and $v = b_1 \ldots b_k$ is the word $a_1 \ldots a_n b_1 \ldots b_k$, denoted uv. The concatenation of u with itself n times is denoted u^n. The set of all words over Σ is denoted by Σ^*. A *language* over Σ is a subset of Σ^*. For example, if $\Sigma = \{a, b\}$, then $\{a^n b^n \mid n \geq 0\}$ is a language over Σ. The concatenation of two languages L and K is $LK = \{uv \mid u \in L, v \in K\}$. L concatenated with itself n times is denoted L^n, and $L^* = \bigcup_{n \geq 0} L^n$.

Finite Automata

In databases, one can model various phenomena using words over some finite alphabet. For example, sequences of database events form words over some alphabet of events. More generally, everything is mapped internally to a sequence of bits, which is nothing but a word over alphabet {0, 1}. The notion of computable query is also formalized using a low-level representation of a database as a word.

An important type of computation over words involves *acceptance*. The objective is to accept precisely the words that belong to some language of interest. The simplest form of acceptance is done using *finite-state automata* (fsa). Intuitively, fsa process words by scanning the word and remembering only a bounded amount of information about what has already been scanned. This is formalized by computation allowing a finite set of states and transitions among the states, driven by the input. Formally, an fsa M over alphabet Σ is a 5-tuple $\langle S, \Sigma, \delta, s_0, F \rangle$, where

- S is a finite set of *states*;
- δ, the *transition function*, is a mapping from $S \times \Sigma$ to S;

- s_0 is a particular state of S, called the *start* state;
- F is a subset of S called the *accepting* states.

An fsa $\langle S, \Sigma, \delta, s_0, F\rangle$ works as follows. The given input word $w = a_1 \ldots a_n$ is read one symbol at a time, from left to right. This can be visualized as a tape on which the input word is written and an fsa with a head that reads symbols from the tape one at a time. The fsa starts in state s_0. One move in state s consists of reading the current symbol a in w, moving to a new state $\delta(s, a)$, and moving the head to the next symbol on the right. If the fsa is in an accepting state after the last symbol in w has been read, w is accepted. Otherwise it is rejected. The language accepted by an fsa M is denoted $L(M)$.

For example, let M be the fsa

$\langle\{\text{even,odd}\}, \{0, 1\}, \delta, \text{even}, \{\text{even}\}\rangle$, with

δ	0	1
even	even	odd
odd	odd	even

The language accepted by M is

$$L(M) = \{w \mid w \text{ has an even number of occurrences of } 1\}.$$

A language accepted by some fsa is called a *regular language*. Not all languages are regular. For example, the language $\{a^n b^n \mid n \geq 0\}$ is not regular. Intuitively, this is so because no fsa can remember the number of a's scanned in order to compare it to the number of b's, if this number is large enough, due to the boundedness of the memory. This property is formalized by the so-called *pumping lemma* for regular languages.

As seen, one way to specify regular languages is by writing an fsa accepting them. An alternative, which is often more convenient, is to specify the shape of the words in the language using so-called regular expressions. A regular expression over Σ is written using the symbols in Σ and the operations concatenation, * and $+$. (The operation $+$ stands for union.) For example, the foregoing language $L(M)$ can be specified by the regular expression $((0^*10^*)^2)^* + 0^*$. To see how regular languages can model things of interest to databases, think of employees who can be affected by the following events:

hire, transfer, quit, fire, retire.

Throughout his or her career, an employee is first hired, can be transferred any number of times, and eventually quits, retires, or is fired. The language whose words are allowable sequences of such events can be specified by a regular expression as *hire (transfer)* (quit + fire + retire)*. One of the nicest features of regular languages is that they have a dual characterization using fsa and regular expressions. Indeed, Kleene's theorem says that a language L is regular iff it can be specified by a regular expression.

There are several important variations of fsa that do not change their accepting power. The first allows scanning the input back and forth any number of times, yielding *two-way* *automata*. The second is *nondeterminism*. A nondeterministic fsa allows several possible next states in a given move. Thus several computations are possible on a given input. A word is accepted if there is at least one computation that ends in an accepting state. Nondeterministic fsa (nfsa) accept the same set of languages as fsa. However, the number of states in the equivalent deterministic fsa may be exponential in the number of states of the nondeterministic one. Thus nondeterminism can be viewed as a convenience allowing much more succinct specification of some regular languages.

Turing Machines and Computability

Turing machines (TMs) provide the classical formalization of computation. They are also used to develop classical complexity theory. Turing machines are like fsa, except that symbols can also be overwritten rather than just read, the head can move in either direction, and the amount of tape available is infinite. Thus a move of a TM consists of reading the current tape symbol, overwriting the symbol with a new one from a specified finite *tape alphabet*, moving the head left or right, and changing state. Like an fsa, a TM can be viewed as an acceptor. The language accepted by a TM M, denoted $L(M)$, consists of the words w such that, on input w, M halts in an accepting state. Alternatively, one can view TM as a generator of words. The TM starts on empty input. To indicate that some word of interest has been generated, the TM goes into some specified state and then continues. Typically, this is a nonterminating computation generating an infinite language. The set of words so generated by some TM M is denoted $G(M)$. Finally, TMs can also be viewed as computing a function from input to output. A TM M computes a partial mapping f from Σ^* to Σ^* if for each $w \in \Sigma^*$: (1) if w is in the domain of f, then M halts on input w with the tape containing the word $f(w)$; (2) otherwise M does not halt on input w.

A function f from Σ^* to Σ^* is *computable* iff there exists some TM computing it. Church's thesis states that any function computable by some reasonable computing device is also computable in the aforementioned sense. So the definition of computability by TMs is robust. In particular, it is insensitive to many variations in the definition of TM, such as allowing multiple tapes. A particularly important variation allows for nondeterminism, similar to nondeterministic fsa. In a nondeterministic TM (NTM), there can be a choice of moves at each step. Thus an NTM has several possible computations on a given input (of which some may be terminating and others not). A word w is accepted by an NTM M if there exists at least one computation of M on w halting in an accepting state.

Another useful variation of the Turing machine is the *counter machine*. Instead of a tape, the counter machine has two stacks on which elements can be pushed or popped. The machine can only test for emptiness of each stack. Counter machines can also define all computable functions. An essentially equivalent and useful formulation of this fact is that the language with integer variables $i, j, \ldots$, two instructions *increment(i)* and *decrement(i)*, and a looping construct *while* $i > 0$ *do*, can define all computable functions on the integers.

Of course, we are often interested in functions on domains other than words—integers are one example. To talk about the computability of such functions on other domains, one goes through an encoding in which each element d of the domain is represented as a word

$enc(d)$ on some fixed, finite alphabet. Given that encoding, it is said that f is computable if the function $enc(f)$ mapping $enc(d)$ to $enc(f(d))$ is computable. This often works without problems, but occasionally it raises tricky issues that are discussed in a few places of this book (particularly in Part E).

It can be shown that a language is $L(M)$ for some acceptor TM M iff it is $G(M)$ for some generator TM M. A language is *recursively enumerable* (r.e.) iff it is $L(M)$ [or $G(M)$] for some TM M. L being r.e. means that there is an algorithm that is guaranteed to say eventually *yes* on input w if $w \in L$ but may run forever if $w \notin L$ (if it stops, it says *no*). Thus one can never know for sure if a word is not in L.

Informally, saying that L is recursive means that there is an algorithm that always decides in finite time whether a given word is in L. If $L = L(M)$ and M always halts, L is *recursive*. A language whose complement is r.e. is called *co-r.e.* The following useful facts can be shown:

1. If L is r.e. and co-r.e., then it is recursive.
2. L is r.e. iff it is the domain of a computable function.
3. L is r.e. iff it is the range of a computable function.
4. L is recursive iff it is the range of a computable nondecreasing function.[1]

As is the case for computability, the notion of recursive is used in many contexts that do not explicitly involve languages. Suppose we are interested in some class of objects called thing-a-ma-jigs. Among these, we want to distinguish widgets, which are those thing-a-ma-jigs with some desirable property. It is said that it is *decidable* if a given thing-a-ma-jig is a widget if there is an algorithm that, given a thing-a-ma-jig, decides in finite time whether the given thing-a-ma-jig is a widget. Otherwise the property is *undecidable*. Formally, thing-a-ma-jigs are encoded as words over some finite alphabet. The property of being a widget is decidable iff the language of words encoding widgets is recursive.

We mention a few classical undecidable problems. The *halting problem* asks if a given TM M halts on a specified input w. This problem is undecidable (i.e., there is no algorithm that, given the description of M and the input w, decides in finite time if M halts on w). More generally it can be shown that, in some precise sense, all nontrivial questions about TMs are undecidable (this is formalized by *Rice's theorem*). A more concrete undecidable problem, which is useful in proofs, is the *Post correspondence problem* (PCP). The input to the PCP consists of two lists

$$u_1, \ldots, u_n; \qquad v_1, \ldots, v_n;$$

of words over some alphabet Σ with at least two symbols. A solution to the PCP is a sequence of indexes $i_1, \ldots, i_k$, $1 \leq i_j \leq n$, such that

$$u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}.$$

[1] f is nondecreasing if $|f(w)| \geq |w|$ for each w.

The question of interest is whether there is a solution to the PCP. For example, consider the input to the PCP problem:

$$\begin{array}{cccc} u_1 & u_2 & u_3 & u_4 \\ aba & bbb & aab & bb \end{array} \qquad \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \\ a & aaa & abab & babba. \end{array}$$

For this input, the PCP has the solution 1, 4, 3, 1; because

$$u_1u_4u_3u_1 = ababbaababa = v_1v_4v_3v_1.$$

Now consider the input consisting of just u_1, u_2, u_3 and v_1, v_2, v_3. An easy case analysis shows that there is no solution to the PCP for this input. In general, it has been shown that it is undecidable whether, for a given input, there exists a solution to the PCP.

The PCP is particularly useful for proving the undecidability of other problems. The proof technique consists of *reducing* the PCP to the problem of interest. For example, suppose we are interested in the question of whether a given thing-a-ma-jig is a widget. The reduction of the PCP to the widget problem consists of finding a computable mapping f that, given an input i to the PCP, produces a thing-a-ma-jig $f(i)$ such that $f(i)$ is a widget iff the PCP has a solution for i. If one can find such a reduction, this shows that it is undecidable if a given thing-a-ma-jig is a widget. Indeed, if this were decidable then one could find an algorithm for the PCP: Given an input i to the PCP, first construct the thing-a-ma-jig $f(i)$, and then apply the algorithm deciding if $f(i)$ is a widget. Because we know that the PCP is undecidable, the property of being a widget cannot be decidable. Of course, any other known undecidable problem can be used in place of the PCP.

A few other important undecidable problems are mentioned in the review of context-free grammars.

Complexity

Suppose a particular problem is solvable. Of course, this does not mean the problem has a *practical* solution, because it may be prohibitively expensive to solve it. Complexity theory studies the difficulty of problems. Difficulty is measured relative to some resources of interest, usually time and space. Again the usual model of reference is the TM. Suppose L is a recursive language, accepted by a TM M that always halts. Let f be a function on positive integers. M is said to use time bounded by f if on every input w, M halts in at most $f(|w|)$ steps. M uses space bounded by f if the amount of tape used by M on every input w is at most $f(|w|)$. The set of recursive languages accepted by TMs using time (space) bounded by f is denoted TIME(f) (SPACE(f)). Let $\mathcal{F}$ be a set of functions on positive integers. Then TIME($\mathcal{F}$) = $\bigcup_{f \in \mathcal{F}}$ TIME(f), and SPACE($\mathcal{F}$) = $\bigcup_{f \in \mathcal{F}}$ SPACE(f). A particularly important class of bounding functions is the polynomials *Poly*. For this class, the following notation has emerged: TIME(*Poly*) is denoted PTIME, and SPACE(*Poly*) is denoted PSPACE. Membership in the class PTIME is often regarded as synonymous to tractability (although, of course, this is not reasonable in all situations, and a case-by-case judgment should be made). Besides the polynomials, it is of interest to consider lower bounds, like logarithmic space. However, because the input itself takes more than logarithmic space to write down, a separation of the input tape from the tape used throughout the computation must be made. Thus the input is given on a read-only tape, and a separate worktape is added. Now let LOGSPACE consist of the recursive languages L that are accepted by some such TM using on input w an amount of worktape bounded by $c \times \log(|w|)$ for some constant c.

Another class of time-bounding functions we shall use is the so-called *elementary* functions. They consist of the set of functions

$$Hyp = \{hyp_i \mid i \geq 0\}, \quad \text{where} \quad \begin{aligned} hyp_0(n) &= n \\ hyp_{i+1}(n) &= 2^{hyp_i(n)}. \end{aligned}$$

The *elementary languages* are those in TIME(*Hyp*).

Nondeterministic TMs can be used to define complexity classes as well. An NTM uses time bounded by f if all computations on input w halt after at most $f(|w|)$ steps. It uses space bounded by f if all computations on input w use at most $f(|w|)$ space (note that termination is not required). The set of recursive languages accepted by some NTM using time bounded by a polynomial is denoted NP, and space bounded by a polynomial is denoted by NPSPACE. Are nondeterministic classes different from their deterministic counterparts? For polynomial space, Savitch's theorem settles the question by showing that PSPACE = NPSPACE (the theorem actually applies to a much more general class of space bounds). For time, things are more complicated. Indeed, the question of whether PTIME equals NP is the most famous open problem in complexity theory. It is generally conjectured that the two classes are distinct.

The following inclusions hold among the complexity classes described:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{NP} \subseteq \text{PSPACE} \subset \text{TIME}(Hyp) = \text{SPACE}(Hyp).$$

All nonstrict inclusions are conjectured to be strict.

Complexity classes of languages can be extended, in the same spirit, to complexity classes of computable functions. Here we look at the resources needed to compute the function rather than just accepting or rejecting the input word.

Consider some complexity class, say $C = \text{TIME}(\mathcal{F})$. Such a class contains all problems that can be solved in time bounded by some function in $\mathcal{F}$. This is an upper bound, so C clearly contains some easy and some hard problems. How can the hard problems be distinguished from the easy ones? This is captured by the notion of *completeness* of a problem in a complexity class. The idea is as follows: A language K in C is complete in C if solving it allows solving all other problems in C, also within C. This is formalized by the notion of *reduction*. Let L and K be languages in C. L is reducible to K if there is a computable mapping f such that for each w, $w \in L$ iff $f(w) \in K$. The definition of reducibility so far guarantees that solving K allows solving L. How about the complexity? Clearly, if the reduction f is hard then we do not have an acceptance algorithm in C. Therefore the complexity of f must be bounded. It might be tempting to use C as the bound. However, this allows all the work of solving L within the reduction, which really makes K irrelevant. Therefore the definition of completeness in a class C requires that the complexity of the reduction function be lower than that for C. More formally, a recursive language is complete in C by C' reductions if for each $L \in C$ there is a function f in C' reducing L to K. The class C' is often understood for some of the main classes C. The conventions we will use are summarized in the following table:

Type of Completeness	*Type of Reduction*
P completeness	LOGSPACE reductions
NP completeness	PTIME reductions
PSPACE completeness	PTIME reductions

Note that to prove that a problem L is complete in C by C' reductions, it is sufficient to exhibit another problem K that is known to be complete in C by C' reductions, and a C' reduction from K to L. Because the C'-reducibility relation is transitive for all customarily used C', it then follows that L is itself C complete by C' reductions. We mention next a few problems that are complete in various classes.

One of the best-known NP-complete problems is the so-called *3-satisfiability (3-SAT)* problem. The input is a propositional formula in conjunctive normal form, in which each conjunct has at most three literals. For example, such an input might be

$$(\neg x_1 \vee \neg x_4 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\neg x_4 \vee x_3 \vee \neg x_1).$$

The question is whether the formula is satisfiable. For example, the preceding formula is satisfied with the truth assignment $\xi(x_1) = \xi(x_2) = \mathit{false}$, $\xi(x_3) = \xi(x_4) = \mathit{true}$. (See Section 2.3 for the definitions of propositional formula and related notions.)

A useful PSPACE-complete problem is the following. The input is a quantified propositional formula (all variables are quantified). The question is whether the formula is true. For example, an input to the problem is

$$\exists x_1 \forall x_2 \forall x_3 \exists x_4 [(\neg x_1 \vee \neg x_4 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\neg x_4 \vee x_3 \vee \neg x_1)].$$

A number of well-known games, such as GO, have been shown to be PSPACE complete.

For PTIME completeness, one can use a natural problem related to context-free grammars (defined next). The input is a context-free grammar G and the question is whether $L(G)$ is empty.

Context-Free Grammars

We have discussed specification of languages using two kinds of acceptors: fsa and TM. Context-free grammars (CFGs) provide different approach to specifying a language that emphasizes the generation of the words in the language rather than acceptance. (Nonetheless, this can be turned into an accepting mechanism by *parsing*.) A CFG is a 4-tuple $\langle N, \Sigma, S, P \rangle$, where

- N is a finite set of *nonterminal symbols*;
- Σ is a finite alphabet of *terminal symbols*, disjoint from N;

- S is a distinguished symbol of N, called the *start symbol*;
- P is a finite set of *productions* of the form $\xi \to w$, where $\xi \in N$ and $w \in (N \cup \Sigma)^*$.

A CFG $G = \langle N, \Sigma, S, P\rangle$ defines a language $L(G)$ consisting of all words in Σ^* that can be *derived* from S by repeated applications of the productions. An application of the production $\xi \to w$ to a word v containing ξ consists of replacing one occurrence of ξ by w. If u is obtained by applying a production to some word v, this is denoted by $u \Rightarrow v$, and the transitive closure of $\Rightarrow$ is denoted $\stackrel{*}{\Rightarrow}$. Thus $L(G) = \{w \mid w \in \Sigma^*, S \stackrel{*}{\Rightarrow} w\}$. A language is called *context free* if it is $L(G)$ for some CFG G. For example, consider the grammar $\langle\{S\}, \{a, b\}, S, P\rangle$, where P consists of the two productions

$$S \to \epsilon,$$
$$S \to aSb.$$

Then $L(G)$ is the language $\{a^n b^n \mid n \geq 0\}$. For example the following is a derivation of a^2b^2:

$$S \Rightarrow aSb \Rightarrow a^2Sb^2 \Rightarrow a^2b^2.$$

The specification power of CFGs lies between that of fsa's and that of TMs. First, all regular languages are context free and all context-free languages are recursive. The language $\{a^n b^n \mid n \geq 0\}$ is context free but not regular. An example of a recursive language that is not context free is $\{a^n b^n c^n \mid n \geq 0\}$. The proof uses an extension to context-free languages of the pumping lemma for regular languages. We also use a similar technique in some of the proofs.

The most common use of CFGs in the area of databases is to view certain objects as CFGs and use known (un)decidability properties about CFGs. Some questions about CFGs known to be decidable are (1) emptiness [is $L(G)$ empty?] and (2) finiteness [is $L(G)$ finite?]. Some undecidable questions are (3) containment [is it true that $L(G_1) \subseteq L(G_2)$?] and (4) equality [is it true that $L(G_1) = L(G_2)$?].

2.3 Basics from Logic

The field of mathematical logic is a main foundation for database theory. It serves as the basis for languages for queries, deductive databases, and constraints. We briefly review the basic notions and notations of mathematical logic and then mention some key differences between this logic in general and the specializations usually considered in database theory. The reader is referred to [EFT84, End72] for comprehensive introductions to mathematical logic, and to the chapter [Apt91] in [Lee91] and [Llo87] for treatments of Herbrand models and logic programming.

Although some previous knowledge of logic would help the reader understand the content of this book, the material is generally self-contained.

Propositional Logic

We begin with the *propositional calculus*. For this we assume an infinite set of *propositional variables*, typically denoted $p, q, r, \ldots$, possibly with subscripts. We also permit the special *propositional constants true* and *false*. (*Well-formed*) *propositional formulas* are constructed from the propositional variables and constants, using the unary connective *negation* ($\neg$) and the binary connectives *disjunction* ($\vee$), *conjunction* ($\wedge$), *implication* ($\rightarrow$), and *equivalence* ($\leftrightarrow$). For example, p, $(p \wedge (\neg q))$ and $((p \vee q) \rightarrow p)$ are well-formed propositional formulas. We generally omit parentheses if not needed for understanding a formula.

A *truth assignment* for a set V of propositional variables is a function $\xi : V \rightarrow \{true, false\}$. The *truth value* $\varphi[\xi]$ of a propositional formula φ under truth assignment ξ for the variables occurring in φ is defined by induction on the structure of φ in the natural manner. For example,

- $true[\xi] = true$;
- if $\varphi = p$ for some variable p, then $\varphi[\xi] = \xi(p)$;
- if $\varphi = (\neg\psi)$ then $\varphi[\xi] = true$ iff $\psi[\xi] = false$;
- $(\psi_1 \vee \psi_2)[\xi] = true$ iff at least one of $\psi_1[\xi] = true$ or $\psi_2[\xi] = true$.

If $\varphi[\xi] = true$ we say that $\varphi[\xi]$ is true and that φ is true under ξ (and similarly for false).

A formula φ is *satisfiable* if there is at least one truth assignment that makes it true, and it is *unsatisfiable* otherwise. It is *valid* if each truth assignment for the variables in φ makes it true. The formula $(p \vee q)$ is satisfiable but not valid; the formula $(p \wedge (\neg p))$ is unsatisfiable; and the formula $(p \vee (\neg p))$ is valid.

A formula φ *logically implies* formula ψ (or ψ is a *logical consequence of* φ), denoted $\varphi \models \psi$ if for each truth assignment ξ, if $\varphi[\xi]$ is true, then $\psi[\xi]$ is true. Formulas φ and ψ are (*logically*) *equivalent*, denoted $\varphi \equiv \psi$, if $\varphi \models \psi$ and $\psi \models \varphi$.

For example, $(p \wedge (p \rightarrow q)) \models q$. Many equivalences for propositional formulas are well known. For example,

$$(\varphi_1 \rightarrow \varphi_2) \equiv ((\neg\varphi_1) \vee \varphi_2); \quad \neg(\varphi_1 \vee \varphi_2) \equiv (\neg\varphi_1 \wedge \neg\varphi_2);$$
$$(\varphi_1 \vee \varphi_2) \wedge \varphi_3 \equiv (\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3); \quad \varphi_1 \wedge \neg\varphi_2 \equiv \varphi_1 \wedge (\varphi_1 \wedge \neg\varphi_2);$$
$$(\varphi_1 \vee (\varphi_2 \vee \varphi_3)) \equiv ((\varphi_1 \vee \varphi_2) \vee \varphi_3).$$

Observe that the last equivalence permits us to view $\vee$ as a polyadic connective. (The same holds for $\wedge$.)

A *literal* is a formula of the form p or $\neg p$ (or *true* or *false*) for some propositional variable p. A propositional formula is in *conjunctive normal form* (CNF) if it has the form $\psi_1 \wedge \cdots \wedge \psi_n$, where each formula ψ_i is a disjunction of literals. *Disjunctive normal form* (DNF) is defined analogously. It is known that if φ is a propositional formula, then there is some formula ψ equivalent to φ that is in CNF (respectively DNF). Note that if φ is in CNF (or DNF), then a shortest equivalent formula ψ in DNF (respectively CNF) may have a length exponential in the length of φ.

First-Order Logic

We now turn to *first-order predicate calculus*. We indicate the main intuitions and concepts underlying first-order logic and describe the primary specializations typically made for database theory. Precise definitions of needed portions of first-order logic are included in Chapters 4 and 5.

First-order logic generalizes propositional logic in several ways. Intuitively, propositional variables are replaced by predicate symbols that range over n-ary relations over an underlying set. Variables are used in first-order logic to range over elements of an abstract set, called the *universe of discourse*. This is realized using the quantifiers $\exists$ and $\forall$. In addition, function symbols are incorporated into the model. The most important definitions used to formalize first-order logic are first-order language, interpretation, logical implication, and provability.

Each first-order language L includes a set of variables, the propositional connectives, the quantifiers $\exists$ and $\forall$, and punctuation symbols ")", "(", and ",". The variation in first-order languages stems from the symbols they include to represent constants, predicates, and functions. More formally, a first-order language includes

(a) a (possibly empty) set of *constant* symbols;
(b) for each $n \geq 0$ a (possibly empty) set of *n-ary predicate symbols*;
(c) for each $n \geq 1$ a (possibly empty) set of *n-ary function symbols*.

In some cases, we also include

(d) the equality symbol $\approx$, which serves as a binary predicate symbol,

and the propositional constants *true* and *false*. It is common to focus on languages that are finite, except for the set of variables.

A familiar first-order language is the language $L_{\mathbf{N}}$ of the nonnegative integers, with

(a) constant symbol $\mathbf{0}$;
(b) binary predicate symbol $\leq$;
(c) binary function symbols $+$, $\times$, and unary $\mathbf{S}$ (successor);

and the equality symbol.

Let L be a first-order language. *Terms* of L are built in the natural fashion from constants, variables, and the function symbols. An *atom* is either *true*, *false*, or an expression of the form $R(t_1, \ldots, t_n)$, where R is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. Atoms correspond to the propositional variables of propositional logic. If the equality symbol is included, then atoms include expressions of the form $t_1 \approx t_2$. The family of (*well-formed predicate calculus*) *formulas* over L is defined recursively starting with atoms, using the Boolean connectives, and using the quantifiers as follows: If φ is a formula and x a variable, then $(\exists x\varphi)$ and $(\forall x\varphi)$ are formulas. As with the propositional case, parentheses are omitted when understood from the context. In addition, $\vee$ and $\wedge$ are viewed as polyadic connectives. A term or formula is *ground* if it involves no variables.

Some examples of formulas in $L_{\mathbf{N}}$ are as follows:

$$\forall x(\mathbf{0} \leq x), \quad \neg(x \approx \mathbf{S}(x)),$$
$$\neg\exists x(\forall y(y \leq x)), \quad \forall y \forall z(x \approx y \times z \rightarrow (y \approx \mathbf{S}(\mathbf{0}) \vee z \approx \mathbf{S}(\mathbf{0}))).$$

(For some binary predicates and functions, we use infix notation.)

The notion of the scope of quantifiers and of *free* and *bound* occurrences of variables in formulas is now defined using recursion on the structure. Each variable occurrence in an atom is free. If φ is $(\psi_1 \vee \psi_2)$, then an occurrence of variable x in φ is free if it is free as an occurrence of ψ_1 or ψ_2; and this is extended to the other propositional connectives. If φ is $\exists y \psi$, then an occurrence of variable $x \neq y$ is free in φ if the corresponding occurrence is free in ψ. Each occurrence of y is bound in φ. In addition, each occurrence of y in φ that is free in ψ is said to be in the *scope* of $\exists y$ at the beginning of φ. A *sentence* is a well-formed formula that has no free variable occurrences.

Until now we have not given a meaning to the symbols of a first-order language and thereby to first-order formulas. This is accomplished with the notion of *interpretation*, which corresponds to the truth assignments of the propositional case. Each interpretation is just one of the many possible ways to give meaning to a language.

An *interpretation* of a first-order language L is a 4-tuple $\mathcal{I} = (U, \mathcal{C}, \mathcal{P}, \mathcal{F})$ where U is a nonempty set of abstract elements called the *universe* (*of discourse*), and $\mathcal{C}$, $\mathcal{P}$, and $\mathcal{F}$ give meanings to the sets of constant symbols, predicate symbols, and function symbols. For example, $\mathcal{C}$ is a function from the constant symbols into U, and $\mathcal{P}$ maps each n-ary predicate symbol p into an n-ary relation over U (i.e., a subset of U^n). It is possible for two distinct constant symbols to map to the same element of U.

When the *equality symbol* denoted $\approx$ is included, the meaning associated with it is restricted so that it enjoys properties usually associated with equality. Two equivalent mechanisms for accomplishing this are described next.

Let $\mathcal{I}$ be an interpretation for language L. As a notational shorthand, if c is a constant symbol in L, we use $c^{\mathcal{I}}$ to denote the element of the universe associated with c by $\mathcal{I}$. This is extended in the natural way to ground terms and atoms.

The usual interpretation for the language $L_{\mathbf{N}}$ is $\mathcal{I}_{\mathbf{N}}$, where the universe is $\mathbf{N}$; $\mathbf{0}$ is mapped to the number 0; $\leq$ is mapped to the usual less than or equal relation; $\mathbf{S}$ is mapped to successor; and $+$ and $\times$ are mapped to addition and multiplication. In such cases, we have, for example, $[\mathbf{S}(\mathbf{S}(\mathbf{0}) + \mathbf{0}))]^{\mathcal{I}_{\mathbf{N}}} \approx 2$.

As a second example related to logic programming, we mention the family of *Herbrand interpretations* of $L_{\mathbf{N}}$. Each of these shares the same universe and the same mappings for the constant and function symbols. An assignment of a universe, and for the constant and function symbols, is called a *preinterpretation*. In the Herbrand preinterpretation for $L_{\mathbf{N}}$, the universe, denoted $U_{L_{\mathbf{N}}}$, is the set containing $\mathbf{0}$ and all terms that can be constructed from this using the function symbols of the language. This is a little confusing because the terms now play a dual role—as terms constructed from components of the language L, and as elements of the universe $U_{L_{\mathbf{N}}}$. The mapping $\mathcal{C}$ maps the constant symbol $\mathbf{0}$ to $\mathbf{0}$ (considered as an element of $U_{L_{\mathbf{N}}}$). Given a term t in U, the function $\mathcal{F}(\mathbf{S})$ maps t to the term $\mathbf{S}(t)$. Given terms t_1 and t_2, the function $\mathcal{F}(+)$ maps the pair (t_1, t_2) to the term $+(t_1, t_2)$, and the function $\mathcal{F}(\times)$ is defined analogously.

The set of ground atoms of $L_{\mathbf{N}}$ (i.e., the set of atoms that do not contain variables) is sometimes called the *Herbrand base* of $L_{\mathbf{N}}$. There is a natural one-one correspondence between interpretations of $L_{\mathbf{N}}$ that extend the Herbrand preinterpretation and subsets of the Herbrand base of $L_{\mathbf{N}}$. One Herbrand interpretation of particular interest is the one that mimics the usual interpretation. In particular, this interpretation maps $\leq$ to the set $\{(t_1, t_2) \mid (t_1^{\mathcal{I}_{\mathbf{N}}}, t_2^{\mathcal{I}_{\mathbf{N}}}) \in \leq^{\mathcal{I}_{\mathbf{N}}}\}$.

We now turn to the notion of satisfaction of a formula by an interpretation. The definition is recursive on the structure of formulas; as a result we need the notion of variable assignment to accommodate variables occurring free in formulas. Let L be a language and $\mathcal{I}$ an interpretation of L with universe U. A *variable assignment* for formula φ is a partial function μ : variables of $L \to U$ whose domain includes all variables free in φ. For terms t, $t^{\mathcal{I},\mu}$ denotes the meaning given to t by $\mathcal{I}$, using μ to interpret the free variables. In addition, if μ is a variable assignment, x is a variable, and $u \in U$, then $\mu[x/u]$ denotes the variable assignment that is identical to μ, except that it maps x to u. We write $I \models \varphi[\mu]$ to indicate that $\mathcal{I}$ *satisfies* φ under μ. This is defined recursively on the structure of formulas in the natural fashion. To indicate the flavor of the definition, we note that $\mathcal{I} \models p(t_1, \ldots, t_n)[\mu]$ if $(t_1^{\mathcal{I},\mu}, \ldots, t_n^{\mathcal{I},\mu}) \in p^{\mathcal{I}}$; $\mathcal{I} \models \exists x \psi[\mu]$ if there is some $u \in U$ such that $\mathcal{I} \models \psi[\mu[x/u]]$; and $\mathcal{I} \models \forall x \psi[\mu]$ if for each $u \in U$, $I \models \psi[\mu[x/u]]$. The Boolean connectives are interpreted in the usual manner. If φ is a sentence, then no variable assignment needs to be specified.

For example, $\mathcal{I}_{\mathbf{N}} \models \forall x \exists y (\neg(x \approx y) \vee x \leq y)$; $\mathcal{I}_{\mathbf{N}} \not\models \mathbf{S}(\mathbf{0}) \leq \mathbf{0}$; and

$$\mathcal{I}_{\mathbf{N}} \models \forall y \forall z (x \approx y \times z \to (y \approx \mathbf{S}(\mathbf{0}) \vee z \approx \mathbf{S}(\mathbf{0})))[\mu]$$

iff $\mu(x)$ is 1 or a prime number.

An interpretation $\mathcal{I}$ is a *model* of a set Φ of sentences if $\mathcal{I}$ satisfies each formula in Φ. The set Φ is *satisfiable* if it has a model.

Logical implication and equivalence are now defined analogously to the propositional case. Sentence φ logically implies sentence ψ, denoted $\varphi \models \psi$, if each interpretation that satisfies φ also satisfies ψ. There are many straightforward equivalences [e.g., $\neg(\neg\varphi) \equiv \varphi$ and $\neg\forall x \varphi \equiv \exists x \neg\varphi$]. Logical implication is generalized to sets of sentences in the natural manner.

It is known that logical implication, considered as a decision problem, is not recursive. One of the fundamental results of mathematical logic is the development of effective procedures for determining logical equivalence. These are based on the notion of *proofs*, and they provide one way to show that logical implication is r.e. One style of proof, attributed to Hilbert, identifies a family of *inference rules* and a family of *axioms*. An example of an inference rule is *modus ponens*, which states that from formulas φ and $\varphi \to \psi$ we may conclude ψ. Examples of axioms are all *tautologies* of propositional logic [e.g., $\neg(\varphi \vee \psi) \leftrightarrow (\neg\varphi \wedge \neg\psi)$ for all formulas φ and ψ], and *substitution* (i.e., $\forall x \varphi \to \varphi_t^x$, where t is an arbitrary term and φ_t^x denotes the formula obtained by simultaneously replacing all occurrences of x free in φ by t). Given a family of inference rules and axioms, a *proof* that set Φ of sentences implies sentence φ is a finite sequence $\psi_0, \psi_1, \ldots, \psi_n = \varphi$, where for each i, either ψ_i is an axiom, or a member of Φ, or it follows from one or more of the previous ψ_j's using an inference rule. In this case we write $\Phi \vdash \varphi$.

The soundness and completeness theorem of Gödel shows that (using *modus ponens* and a specific set of axioms) $\Phi \models \varphi$ iff $\Phi \vdash \varphi$. This important link between $\models$ and $\vdash$ permits the transfer of results obtained in model theory, which focuses primarily on interpretations and models, and proof theory, which focuses primarily on proofs. Notably, a central issue in the study of relational database dependencies (see Part C) has been the search for sound and complete proof systems for subsets of first-order logic that correspond to natural families of constraints.

The model-theoretic and proof-theoretic perspectives lead to two equivalent ways of incorporating equality into first-order languages. Under the model-theoretic approach, the equality predicate $\approx$ is given the meaning $\{(u, u) \mid u \in U\}$ (i.e., normal equality). Under the proof-theoretic approach, a set of *equality axioms* EQ_L is constructed that express the intended meaning of $\approx$. For example, EQ_L includes the sentences $\forall x, y, z(x \approx y \wedge y \approx z \rightarrow x \approx z)$ and $\forall x, y(x \approx y \rightarrow (R(x) \leftrightarrow R(y))$ for each unary predicate symbol R.

Another important result from mathematical logic is the compactness theorem, which can be demonstrated using Gödel's soundness and completeness result. There are two common ways of stating this. The first is that given a (possibly infinite) set of sentences Φ, if $\Phi \models \varphi$ then there is a finite $\Phi' \subseteq \Phi$ such that $\Phi' \models \varphi$. The second is that if each finite subset of Φ is satisfiable, then Φ is satisfiable.

Note that although the compactness theorem guarantees that the Φ in the preceding paragraph has a model, that model is not necessarily finite. Indeed, Φ may only have infinite models. It is of some solace that, among those infinite models, there is surely at least one that is countable (i.e., whose elements can be enumerated: $a_1, a_2, \ldots$). This technically useful result is the Löwenheim-Skolem theorem.

To illustrate the compactness theorem, we show that there is no set Ψ of sentences defining the notion of connectedness in directed graphs. For this we use the language L with two constant symbols, a and b, and one binary relation symbol R, which corresponds to the edges of a directed graph. In addition, because we are working with general first-order logic, both finite and infinite graphs may arise. Suppose now that Ψ is a set of sentences that states that a and b are connected (i.e., that there is a directed path from a to b in R). Let $\Sigma = \{\sigma_i \mid i > 0\}$, where σ_i states "a and b are at least i edges apart from each other." For example, σ_3 might be expressed as

$$\neg R(a, b) \wedge \neg \exists x_1(R(a, x_1) \wedge R(x_1, b)).$$

It is clear that each finite subset of $\Psi \cup \Sigma$ is satisfiable. By the compactness theorem (second statement), this implies that $\Psi \cup \Sigma$ is satisfiable, so it has a model (say, $\mathcal{I}$). In $\mathcal{I}$, there is no directed path between (the elements of the universe identified by) a and b, and so $\mathcal{I} \not\models \Psi$. This is a contradiction.

Specializations to Database Theory

We close by mentioning the primary differences between the general field of mathematical logic and the specializations made in the study of database theory. The most obvious specialization is that database theory has not generally focused on the use of functions on data values, and as a result it generally omits function symbols from the first-order languages used. The two other fundamental specializations are the focus on finite models and the special use of constant symbols.

An interpretation is *finite* if its universe of discourse is finite. Because most databases are finite, most of database theory is focused exclusively on finite interpretations. This is closely related to the field of finite model theory in mathematics.

The notion of logical implication for finite interpretations, usually denoted $\models_{\text{fin}}$, is not equivalent to the usual logical implication $\models$. This is most easily seen by considering the compactness theorem. Let $\Phi = \{\sigma_i \mid i > 0\}$, where σ_i states that there are at least i distinct elements in the universe of discourse. Then by compactness, $\Phi \not\models \mathit{false}$, but by the definition of finite interpretation, $\Phi \models_{\text{fin}} \mathit{false}$.

Another way to show that $\models$ and $\models_{\text{fin}}$ are distinct uses computability theory. It is known that $\models$ is r.e. but not recursive, and it is easily seen that $\models_{\text{fin}}$ is co-r.e. Thus if they were equal, $\models$ would be recursive, a contradiction.

The final specialization of database theory concerns assumptions made about the universe of discourse and the use of constant symbols. Indeed, throughout most of this book we use a fixed, countably infinite set of constants, denoted **dom** (for domain elements). Furthermore, the focus is almost exclusively on finite Herbrand interpretations over **dom**. In particular, for distinct constants c and c', all interpretations that are considered satisfy $\neg c \approx c'$.

Most proofs in database theory involving the first-order predicate calculus are based on model theory, primarily because of the emphasis on finite models and because the link between $\models_{\text{fin}}$ and $\vdash$ does not hold. It is thus informative to identify a mechanism for using traditional proof-theoretic techniques within the context of database theory. For this discussion, consider a first-order language with set **dom** of constant symbols and predicate symbols $R_1, \ldots, R_n$. As will be seen in Chapter 3, a database *instance* is a finite Herbrand interpretation **I** of this language. Following [Rei84], a family $\Sigma_{\mathbf{I}}$ of sentences is associated with **I**. This family includes the axioms of equality (mentioned earlier) and

Atoms: $R_i(\vec{a})$ for each $\vec{a}$ in $R_i^{\mathbf{I}}$.

Extension axioms: $\forall \vec{x}(R_i(\vec{x}) \leftrightarrow (\vec{x} \approx \vec{a}_1 \vee \cdots \vee \vec{x} \approx \vec{a}_m))$, where $\vec{a}_1, \ldots, \vec{a}_m$ is a listing of all elements of $R_i^{\mathbf{I}}$, and we are abusing notation by letting $\approx$ range over vectors of terms.

Unique Name axioms: $\neg c \approx c'$ for each distinct pair c, c' of constants occurring in **I**.

Domain Closure axiom: $\forall x(x \approx c_1 \vee \cdots \vee x \approx c_n)$, where $c_1, \ldots, c_n$ is a listing of all constants occurring in **I**.

A set of sentences obtained in this manner is termed an *extended relational theory*.

The first two sets of sentences of an extended relational theory express the specific contents of the relations (predicate symbols) of **I**. Importantly, the Extension sentences ensure that for any (not necessarily Herbrand) interpretation $\mathcal{J}$ satisfying $\Sigma_{\mathbf{I}}$, an n-tuple is in $R_i^{\mathcal{J}}$ iff it equals one of the n-tuples in $R_i^{\mathbf{I}}$. The Unique Name axiom ensures that no pair of distinct constants is mapped to the same element in the universe of $\mathcal{J}$, and the Domain Closure axiom ensures that each element of the universe of $\mathcal{J}$ equals some constant occurring in **I**. For all intents and purposes, then, any interpretation $\mathcal{J}$ that models $\Sigma_{\mathbf{I}}$ is isomorphic to **I**, modulo condensing under equivalence classes induced by $\approx^{\mathcal{J}}$. Importantly, the following link with conventional logical implication now holds: For any set Γ of sentences, $\mathbf{I} \models \Gamma$ iff $\Sigma_{\mathbf{I}} \cup \Gamma$ is satisfiable. The perspective obtained through this connection with classical logic is useful when attempting to extend the conventional relational model (e.g., to incorporate so-called incomplete information, as discussed in Chapter 19).

The Extension axioms correspond to the intuition that a tuple $\vec{a}$ is in relation R only if it is explicitly included in R by the database instance. A more general formulation of this intuition is given by the *closed world assumption* (CWA) [Rei78]. In its most general formulation, the CWA is an inference rule that is used in proof-theoretic contexts. Given a set Σ of sentences describing a (possibly nonconventional) database instance, the CWA states that one can infer a negated atom $R(\vec{a})$ if $\Sigma \nvdash R(\vec{a})$ [i.e., if one cannot prove $R(\vec{a})$ from Σ using conventional first-order logic]. In the case where Σ is an extended relational theory this gives no added information, but in other contexts (such as deductive databases) it does. The CWA is related in spirit to the *negation as failure* rule of [Cla78].