Contents

1 Introduction 1
- 1.1 A Database Example 1
- 1.2 An Example from Complexity Theory 4
- 1.3 An Example from Formal Language Theory 6
- 1.4 An Overview of the Book 8
- 1.5 Exercises 10

2 Preliminaries 13
- 2.1 Background from Mathematical Logic 13
- 2.2 Background from Automata and Computability Theory 17
- 2.3 Background from Complexity Theory 19
- 2.4 Bibliographic Notes 21

3 Ehrenfeucht-Fraïssé Games 23
- 3.1 First Inexpressibility Proofs 23
- 3.2 Definition and Examples of Ehrenfeucht-Fraïssé Games 26
- 3.3 Games and the Expressive Power of FO 32
- 3.4 Rank-k Types 33
- 3.5 Proof of the Ehrenfeucht-Fraïssé Theorem 35
- 3.6 More Inexpressibility Results 37
- 3.7 Bibliographic Notes 40
- 3.8 Exercises 41

4 Locality and Winning Games 45
- 4.1 Neighborhoods, Hanf-locality, and Gaifman-locality 45
- 4.2 Combinatorics of Neighborhoods 49
- 4.3 Locality of FO 51
- 4.4 Structures of Small Degree 54
- 4.5 Locality of FO Revisited 57
- 4.6 Bibliographic Notes 62
- 4.7 Exercises 63

5 Ordered Structures 67
5.1 Invariant Queries 67
5.2 The Power of Order-invariant FO 69
5.3 Locality of Order-invariant FO 73
5.4 Bibliographic Notes 83
5.5 Exercises 83

6 Complexity of First-Order Logic 87
6.1 Data, Expression, and Combined Complexity 87
6.2 Circuits and FO Queries 89
6.3 Expressive Power with Arbitrary Predicates 93
6.4 Uniformity and AC^0 95
6.5 Combined Complexity of FO 99
6.6 Parametric Complexity and Locality 99
6.7 Conjunctive Queries 102
6.8 Bibliographic Notes 108
6.9 Exercises 109

7 Monadic Second-Order Logic and Automata 113
7.1 Second-Order Logic and Its Fragments 113
7.2 MSO Games and Types 116
7.3 Existential and Universal MSO on Graphs 119
7.4 MSO on Strings and Regular Languages 124
7.5 FO on Strings and Star-Free Languages 127
7.6 Tree Automata 129
7.7 Complexity of MSO 133
7.8 Bibliographic Notes 136
7.9 Exercises 137

8 Logics with Counting 141
8.1 Counting and Unary Quantifiers 141
8.2 An Infinitary Counting Logic 145
8.3 Games for $\mathcal{L}^*_{\infty\omega}(\mathbf{Cnt})$ 151
8.4 Counting and Locality 153
8.5 Complexity of Counting Quantifiers 155
8.6 Aggregate Operators 158
8.7 Bibliographic Notes 161
8.8 Exercises 161

9 Turing Machines and Finite Models 165
9.1 Trakhtenbrot's Theorem and Failure of Completeness 165
9.2 Fagin's Theorem and NP 168
9.3 Bibliographic Notes 174
9.4 Exercises 174

10 Fixed Point Logics and Complexity Classes 177
10.1 Fixed Points of Operators on Sets 178
10.2 Fixed Point Logics 180
10.3 Properties of LFP and IFP 184
10.4 LFP, PFP, and Polynomial Time and Space 192
10.5 DATALOG and LFP 195
10.6 Transitive Closure Logic 199
10.7 A Logic for PTIME? 204
10.8 Bibliographic Notes 206
10.9 Exercises 207

11 Finite Variable Logics 211
11.1 Logics with Finitely Many Variables 211
11.2 Pebble Games 215
11.3 Definability of Types 220
11.4 Ordering of Types 225
11.5 Canonical Structures and the Abiteboul-Vianu Theorem 229
11.6 Bibliographic Notes 232
11.7 Exercises 233

12 Zero-One Laws 235
12.1 Asymptotic Probabilities and Zero-One Laws 235
12.2 Extension Axioms 238
12.3 The Random Graph 241
12.4 Zero-One Law and Second-Order Logic 243
12.5 Almost Everywhere Equivalence of Logics 245
12.6 Bibliographic Notes 246
12.7 Exercises 247

13 Embedded Finite Models 249
13.1 Embedded Finite Models: the Setting 249
13.2 Analyzing Embedded Finite Models 252
13.3 Active-Generic Collapse 256
13.4 Restricted Quantifier Collapse 260
13.5 The Random Graph and Collapse to MSO 265
13.6 An Application: Constraint Databases 267
13.7 Bibliographic Notes 270
13.8 Exercises 271

14 Other Applications of Finite Model Theory 275
14.1 Finite Model Property and Decision Problems 275
14.2 Temporal and Modal Logics 278
14.3 Constraint Satisfaction and Homomorphisms of Finite Models 285
14.4 Bibliographic Notes 288

References 291

List of Notation 305

Index 307

Name Index 313

1

Introduction

Finite model theory studies the expressive power of logics on finite models. Classical model theory, on the other hand, concentrates on infinite structures: its origins are in mathematics, and most objects of interest in mathematics are infinite, e.g., the sets of natural numbers, real numbers, etc. Typical examples of interest to a model-theorist would be algebraically closed fields (e.g., $\langle \mathbb{C}, +, \cdot \rangle$), real closed fields (e.g., $\langle \mathbb{R}, +, \cdot, < \rangle$), various models of arithmetic (e.g., $\langle \mathbb{N}, +, \cdot \rangle$ or $\langle \mathbb{N}, + \rangle$), and other structures such as Boolean algebras or random graphs.

The origins of finite model theory are in computer science where most objects of interest are finite. One is interested in the expressiveness of logics over finite graphs, or finite strings, other finite relational structures, and sometimes restrictions of arithmetic structures to an initial segment of natural numbers.

The areas of computer science that served as a primary source of examples, as well as the main consumers of techniques from finite model theory, are databases, complexity theory, and formal languages (although finite model theory found applications in other areas such as AI and verification). In this chapter, we give three examples that illustrate the need for studying logics over finite structures.

1.1 A Database Example

While early database systems used rather ad hoc data models, from the early 1970s the world switched to the relational model. In that model, a database stores tables, or relations, and is queried by a logic-based declarative language. The most standard such language, *relational calculus*, has precisely the power of first-order predicate calculus. In real life, it comes equipped with a specialized programming syntax (e.g., the `select-from-where` statement of SQL).

Suppose that we have a company database, and one of its relations is the `Reports_To` relation: it stores pairs (x, y), where x is an employee, and y is

his/her immediate manager. Organizational hierarchies tend to be quite complicated and often result in many layers of management, so one may want to skip the immediate manager level and instead look for the manager's manager.

In SQL, this would be done by the following query:

```
select R1.employee, R2.manager
from   Reports_To R1, Reports_To R2
where  R1.manager=R2.employee
```

This is simply a different way of writing the following first-order logic formula:

$$\varphi(x,y) \quad \equiv \quad \exists z \left(\texttt{Reports_To}(x,z) \wedge \texttt{Reports_To}(z,y) \right) .$$

Continuing, we may ask for someone's manager's manager's manager:

$$\exists z_1 \exists z_2 \left(\texttt{Reports_To}(x,z_1) \wedge \texttt{Reports_To}(z_1,z_2) \wedge \texttt{Reports_To}(z_2,y) \right) ,$$

and so on.

But what if we want to find everyone who is higher in the hierarchy than a given employee? Speaking graph-theoretically, if we associate a pair (x,y) in the `Reports_To` relation with a directed edge from x to y in a graph, then we want to find, for a given node, all the nodes reachable from it. This does not seem possible in first-order logic, but how can one prove this?

There are other queries naturally related to this reachability property. Suppose that once in a while, the company wants to make sure that its management hierarchy is logically consistent; that is, we cannot have cycles in the `Reports_To` relation. In graph-theoretic terms, it means that `Reports_To` is acyclic. Again, if one thinks about it for a while, it seems that first-order logic does not have enough power to express this query.

We now consider a different kind of query. Suppose we have two managers, x and y, and let X be the set of all the employees directly managed by x (i.e., all x' such that (x',x) is in `Reports_To`), and likewise let Y be the set of all the employees directly managed by y. Can we write a query asking whether $|X| = |Y|$; that is, a query asking whether x and y have the same number of people reporting to them?

It turns out that first-order logic is again not sufficiently expressive for this kind of query, but since queries like those described above are so common in practice, SQL adds special features to the language to perform them. That is, SQL can count: it can apply the cardinality function (and more complex functions as well) to entire columns in relations. For example, in SQL one can write a query that finds all pairs of managers x and y who have the same number of people reporting to them:

```
select R1.manager, R2.manager
from   Reports_To R1, Reports_To R2
where   (select count(Reports_To.employee)
         from Reports_To
         where Reports_To.manager = R1.manager)
      = (select count(Reports_To.employee)
         from Reports_To
         where Reports_To.manager = R2.manager)
```

Since this cannot be done in first-order logic, but can be done in SQL (and, in fact, in some rather simple extensions of first-order logic with counting), it is natural to ask whether counting provides enough expressiveness to define queries such as reachability (can node x be reached from node y in a given graph?) and acyclicity.

Typical applications of finite model theory in databases have to deal with questions of this sort: what can, and, more importantly, what cannot, be expressed in various query languages.

Let us now give intuitive reasons why reachability queries are not expressible in first-order logic. Consider a different example. Suppose that we have an airline database, with a binary relation R (for routes), such that an entry (A, B) in R indicates that there is a flight from A to B. Now suppose we want to find all pairs of cities A, B such that there is a direct flight between them; this is done by the following query:

$$q_0(x, y) \quad \equiv \quad R(x, y),$$

which is simply a first-order formula with two free variables. Next, suppose we want to know if one can get from x to y with exactly one change of plane; then we write

$$q_1(x, y) \quad \equiv \quad \exists z \; R(x, z) \wedge R(z, y).$$

Doing "with at most one change" means having a disjunction

$$Q_1(x, y) \quad \equiv \quad q_1(x, y) \vee q_0(x, y).$$

Clearly, for each fixed k we can write a formula stating that one can get from x to y with exactly k stops:

$$q_k(x, y) \quad \equiv \quad \exists z_1 \ldots \exists z_k \; R(x, z_1) \wedge R(z_1, z_2) \wedge \ldots \wedge R(z_k, y),$$

as well as $Q_k = \bigvee_{j \leq k} q_j$ testing if at most k stops suffice.

But what about the *reachability* query: can we get from x to y? That is, one wants to compute the transitive closure of R. The problem with this is that we do not know in advance what k is supposed to be. So the query that we need to write is

$$\bigvee_{k \in \mathbb{N}} q_k,$$

but this is not a first-order formula! Of course this is not a formal proof that reachability is not expressible in first-order logic (we shall see a proof of this fact in Chap. 3), but at least it gives a hint as to what the limitations of first-order logic are.

The inability of first-order logic to express some important queries motivated a lot of research on extensions of first-order logic that can do queries such as transitive closure or cardinality comparisons. We shall see a number of extensions of these kinds – fixed point logics, (fragments of) second-order logic, counting logics – that are important for database theory, and we shall study properties of these extensions as well.

1.2 An Example from Complexity Theory

We now turn to a different area, and to more expressive logics. Suppose that we have a graph, this time *undirected*, given to us as a pair $\langle V, E\rangle$, where V is the set of vertices, or nodes, and E is the edge relation. Assume that now we can specify graph properties in *second-order logic*; that is, we can quantify over *sets* (or relations) of nodes.

Consider a well-known property of *Hamiltonicity*. A *simple circuit* in a graph G is a sequence $(a_1, \ldots, a_n)$ of distinct nodes such that there are edges $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n), (a_n, a_1)$. A simple circuit is Hamiltonian if $V = \{a_1, \ldots, a_n\}$. A graph is *Hamiltonian* if it has a Hamiltonian circuit.

We now consider the following formula:

$$\exists L\ \exists S \left(\begin{array}{l} \quad \textit{linear order}(L) \\ \wedge\ S \textit{ is the successor relation of } L \\ \wedge\ \forall x \exists y\ (L(x,y) \vee L(y,x)) \\ \wedge\ \forall x \forall y\ (S(x,y) \rightarrow E(x,y)) \end{array} \right) \tag{1.1}$$

The quantifiers $\exists L\ \exists S$ state the existence of two binary relations, L and S, that satisfy the formula in parentheses. That formula uses some abbreviations. The subformula *linear order*(L) in (1.1) states that the relation L is a linear ordering; it can be defined as

$$\begin{array}{rl} & \big(\forall x \neg L(x,x)\big)\ \wedge\ \big(\forall x \forall y \forall z\ (L(x,y) \wedge L(y,z) \rightarrow L(x,z))\big) \\ \wedge & \forall x \forall y \Big((x \neq y) \rightarrow \big(L(x,y) \vee L(y,x)\big)\Big). \end{array}$$

The subformula S *is the successor relation of* L states that S is the successor relation associated with the linear ordering L; it can be defined as

$$\forall x \forall y\ \ S(x,y) \leftrightarrow \left(\begin{array}{l} \Big(L(x,y) \wedge \neg \exists z \big(L(x,z) \wedge L(z,y)\big)\Big) \\ \vee\ \big(\neg \exists z\ L(x,z)\ \wedge\ \neg \exists z\ L(z,y)\big) \end{array} \right)$$

Note that S is the circular successor relation, as it also includes the pair (x, y) where x is the maximal and y the minimal element with respect to L.

Then (1.1) says that L and S are defined on all nodes of the graph, and that S is a subset of E. Hence, S is a Hamiltonian circuit, and thus (1.1) tests if a graph is Hamiltonian.

It it well known that testing Hamiltonicity is an NP-complete problem. Is this a coincidence, or is there a natural connection between NP and second-order logic? Let us turn our attention to two other well-known NP-complete problems: *3-colorability* and *clique*.

To test if a graph is 3-colorable, we have to check that there exist three disjoint sets A, B, C covering the nodes of the graph such that for every edge $(a, b) \in E$, the nodes a and b cannot belong to the same set. The sentence below does precisely that:

$$\exists A \exists B \exists C \left(\begin{array}{l} \forall x \left[\begin{array}{l} (A(x) \wedge \neg B(x) \wedge \neg C(x)) \\ \vee\ (\neg A(x) \wedge B(x) \wedge \neg C(x)) \\ \vee\ (\neg A(x) \wedge \neg B(x) \wedge C(x)) \end{array} \right] \\ \wedge \\ \forall x, y\ E(x,y) \rightarrow \neg \left[\begin{array}{l} (A(x) \wedge A(y)) \\ \vee\ (B(x) \wedge B(y)) \\ \vee\ (C(x) \wedge C(y)) \end{array} \right] \end{array} \right) \tag{1.2}$$

For *clique*, typically one has a parameter k, and the problem is to check whether a clique of size k exists. Here, to stay purely within the formalism of second-order logic, we assume that the input is a graph E and a set of nodes (a unary relation) U, and we ask if E has a clique of size $|\,U\,|$. We do it by testing if there is a set C (nodes of the clique) and a binary relation F that is a one-to-one correspondence between C and U. Testing that the restriction of E to C is a clique, and that F is one-to-one, can be done in first-order logic. Thus, the test is done by the following second-order sentence:

$$\exists C \exists F \left(\begin{array}{l} \forall x \forall y\ \big(F(x,y) \rightarrow (C(x) \wedge U(y))\big) \\ \wedge\ \forall x\ \big(C(x)\ \rightarrow\ \exists! y (F(x,y) \wedge U(y))\big) \\ \wedge\ \forall y\ \big(U(y) \rightarrow \exists! x (F(x,y) \wedge C(x))\big) \\ \wedge\ \forall x \forall y\ \big(C(x) \wedge C(y) \rightarrow E(x,y)\big) \end{array} \right) \tag{1.3}$$

Here $\exists! x \varphi(x)$ means "there exists exactly one x such that $\varphi(x)$"; this is an abbreviation for $\exists x \big(\varphi(x)\ \wedge\ \forall y\ (\varphi(y) \rightarrow x = y)\big)$.

Notice that (1.1), (1.2), and (1.3) all follow the same pattern: they start with existential second-order quantifiers, followed by a first-order formula. Such formulas form what is called *existential second-order logic*, abbreviated as $\exists$SO. The connection to NP can easily be seen: existential second-order quantifiers correspond to the guessing stage of an NP algorithm, and the remaining first-order formula corresponds to the polynomial time verification stage of an NP algorithm.

It turns out that the connection between NP and ∃SO is exact, as was shown by Fagin in his celebrated 1974 theorem, stating that NP = ∃SO. This connection opened up a new area, called descriptive complexity. The goals of descriptive complexity are to describe complexity classes by means of logical formalisms, and then use tools from mathematical logic to analyze those classes. We shall prove Fagin's theorem later, and we shall also see logical characterizations of a number of other familiar complexity classes.

1.3 An Example from Formal Language Theory

Now we turn our attention to strings over a finite alphabet, say $\Sigma = \{a, b\}$. We want to represent a string as a structure, much like a graph.

Given a string $s = s_1 s_2 \dots s_n$, we create a structure M_s as follows: the universe is $\{1, \dots, n\}$ (corresponding to positions in the string), we have one binary relation $<$ whose meaning of course is the usual order on the natural numbers, and two unary relations A and B. Then $A(i)$ is true if $s_i = a$, and $B(i)$ is true if $s_i = b$. For example, M_{abba} has universe $\{1, 2, 3, 4\}$, with A interpreted as $\{1, 4\}$ and B as $\{2, 3\}$.

Let us look at the following second-order sentence in which quantifiers range over sets of positions in a string:

$$\Phi \quad \equiv \quad \exists X \exists Y \begin{pmatrix} \forall x \ \big(X(x) \leftrightarrow \neg Y(x)\big) \\ \wedge \ \forall x \ \forall y \ (X(x) \wedge Y(y) \rightarrow x < y) \\ \wedge \ \forall x \ (X(x) \rightarrow A(x) \wedge Y(x) \rightarrow B(x)) \end{pmatrix}$$

When is M_s a model of Φ? This happens iff there exists two sets of positions, X and Y, such that X and Y form a partition of the universe (this is what the first conjunct says), that all positions in X precede the positions in Y (that is what the second conjunct says), and that for each position i in X, the ith symbol of s is a, for each position j in Y, the jth symbol is b (this is stated in the third conjunct). That is, the string starts with some a's, and then switches to all b's. Using the language of regular expressions, we can say that

$$M_s \models \Phi \quad \text{iff} \quad s \in a^* b^*.$$

Is quantification over sets really necessary in this example? It turns out that the answer is *no*: one can express the fact that s is in a^*b^* by saying that there are no two positions $i < j$ such that the ith symbol is b and the jth symbol is a. This, of course, can be done in first-order logic:

$$\neg \exists i \exists j \Big((i < j) \ \wedge \ B(i) \ \wedge \ A(j) \Big).$$

A natural question that arises then is the following: are second-order quantifiers of no use if one wants to describe regular languages by logical means? The answer is *no*, as we shall see later. For now, we can give an example.

First, consider the sentence $\Phi_a \equiv \forall i\ A(i)$, which is true in M_s iff $s \in a^*$. Next, define a relation $i \prec j$ saying that j is the successor of i. It can be defined by the formula $\big((i < j) \wedge \forall k\ ((k \leq i) \vee (k \geq j))\big)$. Now consider the sentence

$$\Phi_1 \quad \equiv \quad \exists X \exists Y \begin{pmatrix} & \forall i\ \big(X(i) \leftrightarrow \neg Y(i)\big) \\ \wedge & \forall i\ \big(\neg\exists j(j < i) \rightarrow X(i)\big) \\ \wedge & \forall i\ \big(\neg\exists j(j > i) \rightarrow Y(i)\big) \\ \wedge & \forall i \forall j\ \big((i \prec j) \wedge X(i) \rightarrow Y(j)\big) \\ \wedge & \forall i \forall j\ \big((i \prec j) \wedge Y(i) \rightarrow X(j)\big) \end{pmatrix}$$

This sentence says that the universe $\{1, \ldots, n\}$ can be partitioned into two sets X and Y such that $1 \in X$, $n \in Y$, and the successor of an element of X is in Y and vice versa; that is, the size of the universe is even.

Now what is $\Phi_1 \wedge \Phi_a$? It says that the string is of even length, and has only a's in it – hence, $M_s \models \Phi_1 \wedge \Phi_a$ iff $s \in (aa)^*$. It turns out that one cannot define $(aa)^*$ using first-order logic alone: one needs second-order quantifiers. Moreover, with second-order quantifiers ranging over sets of positions, one defines *precisely* the regular languages. We shall deal with both expressibility and inexpressibility results related to logics over strings later in this book.

There are a number of common themes in the examples presented above. In all the cases, we are talking about the expressive power of logics over finite objects: relational databases, graphs, and strings. There is a close connection between logical formalisms and familiar concepts from computer science: first-order logic corresponds to relational calculus, existential second-order logic to the complexity class NP, and second-order logic with quantifiers ranging over sets describes regular languages.

Of equal importance is the fact that in all the examples we want to show some *inexpressibility* results. In the database example, we want to show that the transitive closure is not expressible in first-order logic. In the complexity example, it would be nice to show that certain problems cannot be expressed in $\exists$SO – any such result would give us bounds on the class NP, and this would hopefully lead to separation results for complexity classes. In the example from formal languages, we want to show that certain regular languages (e.g., $(aa)^*$) cannot be expressed in first-order logic.

Inexpressibility results have traditionally been a core theme of finite model theory. The main explanation for that is the source of motivating examples for finite model theory. Most of them come from computer science, where one is dealing not with natural phenomena, but rather with artificial creations. Thus, we often want to know the limitations of these creations. In general, this explains the popularity of impossibility results in computer science. After all, the most famous open problem of computer science, the PTIME vs NP problem, is so fascinating because the expected answer would tell us that a large number of important problems *cannot* be solved efficiently.

Concentrating on inexpressibility results highlights another important feature of finite model theory: since we are often interested in counterexamples, many constructions and techniques of interest apply only to a "small" fraction of structures. In fact, we shall see that some techniques (e.g., locality) degenerate to trivial statements on almost all structures, and yet it is that small fraction of structures on which they behave interestingly that gives us important techniques for analyzing expressiveness of logics, query languages, etc. Towards the end of the book, we shall also see that on most typical structures, some very expressive logics collapse to rather weak ones; however, all interesting separation examples occur outside the class of "typical" structures.

1.4 An Overview of the Book

In Chap. 2, we review the background material from mathematical logic, computability theory, and complexity theory.

In Chap. 3 we introduce the fundamental tool of Ehrenfeucht-Fraïssé games, and prove their completeness for expressibility in first-order logic (FO). The game is played by two players, the spoiler and the duplicator, on two structures. The spoiler tries to show that the structures are different, while the duplicator tries to show that they are the same. If the duplicator can succeed for k rounds of such a game, it means that the structures cannot be distinguished by FO sentences whose depth of quantifier nesting does not exceed k. We also define types, which play a very important role in many aspects of finite model theory. In the same chapter, we see some bounds on the expressive power of FO, proved via Ehrenfeucht-Fraïssé games.

Finding winning strategies in Ehrenfeucht-Fraïssé games becomes quite hard for nontrivial structures. Thus, in Chap. 4, we introduce some sufficient conditions that guarantee a win for the duplicator. These conditions are based on the idea of locality. Intuitively, local formulae cannot see very far from their free variables. We show several different ways of formalizing this intuition, and explain how each of those ways gives us easy proofs of bounds on the expressiveness of FO.

In Chap. 5 we continue to study first-order logic, but this time over structures whose universe is ordered. Here we see the phenomenon that is very common for logics over finite structures. We call a property of structures order-invariant if it can be defined with a linear order, but is independent of a particular linear order used. It turns out that there are order-invariant FO-definable properties that are not definable in FO alone. We also show that such order-invariant properties continue to be local.

Chap. 6 deals with the complexity of FO. We distinguish two kinds of complexity: data complexity, meaning that a formula is fixed and the structure on which it is evaluated varies, and combined complexity, meaning that both the formula and the structure are part of the input. We show how to evaluate

FO formulae by Boolean circuits, and use this to derive drastically different bounds for the complexity of FO: AC^0 for data complexity, and PSPACE for combined complexity. We also consider the parametric complexity of FO: in this case, the formula is viewed as a parameter of the input. Finally, we study a subclass of FO queries, called conjunctive queries, which is very important in database theory, and prove complexity bounds for it.

In Chap. 7, we move away from FO, and consider its extension with monadic second-order quantifiers: such quantifiers can range over subsets of the universe. The resulting logic is called monadic second-order logic, or MSO. We also consider two restrictions of MSO: an $\exists$MSO formula starts with a sequence of existential second-order quantifiers, which is followed by an FO formula, and an $\forall$MSO formula starts with a sequence of universal second-order quantifiers, followed by an FO formula. We first study $\exists$MSO and $\forall$MSO on graphs, where they are shown to be different. We then move to strings, where MSO collapses to $\exists$MSO and captures precisely the regular languages. Further restricting our attention to FO over strings, we prove that it captures the star-free languages. We also cover MSO over trees, and tree automata.

In Chap. 8 we study a different extension of FO: this time, we add mechanisms for counting, such as counting terms, counting quantifiers, or certain generalized unary quantifiers. We also introduce a logic that has a lot of counting power, and prove that it remains local, much as FO. We apply these results in the database setting, considering a standard feature of many query languages – aggregate functions – and proving bounds on the expressiveness of languages with aggregation.

In Chap. 9 we present the technique of coding Turing machines as finite structures, and use it to prove two results: Trakhtenbrot's theorem, which says that the set of finitely satisfiable sentences is not recursive, and Fagin's theorem, which says that NP problems are precisely those expressible in existential second-order logic.

Chapter 10 deals with extensions of FO for expressing properties that, algorithmically, require recursion. Such extensions have fixed point operators. There are three flavors of them: least, inflationary, and partial fixed point operators. We study properties of resulting fixed point logics, and prove that in the presence of a linear order, they capture complexity classes PTIME (for least and inflationary fixed points) and PSPACE (for partial fixed points). We also deal with a well-known database query language that adds fixed points to FO: DATALOG. In the same chapter, we consider a closely related logic based on adding the transitive closure operator to FO, and prove that over order structures it captures nondeterministic logarithmic space.

Fixed point logics are not very easy to analyze. Nevertheless, they can be embedded into a logic which uses infinitary connectives, but has a restriction that every formula only mentions finitely many variables. This logic, and its fragments, are studied in Chap. 11. We introduce the logic $\mathcal{L}^{\omega}_{\infty\omega}$, define games for it, and prove that fixed point logics are embeddable into it. We

study definability of types for finite variable logics, and use them to provide a purely logical counterpart of the PTIME vs. PSPACE question.

In Chap. 12 we study the asymptotic behavior of FO and prove that every FO sentence is either true in almost all structures, or false in almost all structures. This phenomenon is known as the zero-one law. We also prove that $\mathcal{L}^{\omega}_{\infty\omega}$, and hence fixed point logics, have the zero-one law. In the same chapter we define an infinite structure whose theory consists precisely of FO sentences that hold in almost all structures. We also prove that almost everywhere, fixed point logics collapse to FO.

In Chap. 13, we show how finite and infinite model theory mix: we look at finite structures that live in an infinite one, and study the power of FO over such hybrid structures. We prove that for some underlying infinite structures, like $\langle \mathbb{N}, +, \cdot \rangle$, every computable property of finite structures embedded into them can be defined, but for others, like $\langle \mathbb{R}, +, \cdot \rangle$, one can only define properties which are already expressible in FO over the finite structure alone. We also explain connections between such mixed logics and database query languages.

Finally, in Chap. 14, we outline other applications of finite model theory: in decision problems in mathematical logic, in formal verification of properties of finite state systems, and in constraint satisfaction.

1.5 Exercises

Exercise 1.1. Show how to express the following properties of graphs in first-order logic:

- A graph is complete.
- A graph has an isolated vertex.
- A graph has at least two vertices of out-degree 3.
- Every vertex is connected by an edge to a vertex of out-degree 3.

Exercise 1.2. Show how to express the following properties of graphs in existential second-order logic:

- A graph has a kernel, i.e., a set of vertices X such that there is no edge between any two vertices in X, and every vertex outside of X is connected by an edge to a vertex of X.
- A graph on n vertices has an independent set X (i.e., no two nodes in X are connected by an edge) of size at least $n/2$.
- A graph has an even number of vertices.
- A graph has an even number of edges.
- A graph with m edges has a bipartite subgraph with at least $m/2$ edges.

Exercise 1.3. (a) Show how to define the following regular languages in monadic second-order logic:

- $a^*(b+c)^*aa^*$;

- $(aaa)^*(bb)^+$;
- $\Big(\big((a+b)^*cc^*\big)^*(aa)^*\Big)^* a$.

For the first language, provide a first-order definition as well.

(b) Let Φ be a monadic second-order logic sentence over strings. Show how to construct a sentence Ψ such that $M_s \models \Psi$ iff there is a string s' such that $|s|=|s'|$ and $M_{s \cdot s'} \models \Phi$. Here $|s|$ refers to the length of s, and $s \cdot s'$ is the concatenation of s and s'.

Remark: once we prove Büchi's theorem in Chap. 7, you will see that the above statement says that if L is a regular language, then the language

$$\frac{1}{2}L \;=\; \{s \mid \text{for some } s',\ |s|=|s'| \text{ and } s \cdot s' \in L\}$$

is regular too (see, e.g., Exercise 3.16 in Hopcroft and Ullman [126]).