UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2015:16

Undecidability of finite satisfiability and characterization of NP in finite model theory

Max Block

Examensarbete i matematik, 15 hp
Handledare och examinator: Vera Koponen
Juni 2015

Department of Mathematics
Uppsala University

Abstract

This degree project is about two fundamental results in finite model theory, which is an area of mathematical logic with applications in computer science. Usually the structures of interest for computer scientists may be regarded as finite models for some formal language.

One of the first results, sometimes regarded as the birth of finite model theory, is **Trakhtenbrot's** result from 1950 stating that validity over finite models is not recursively enumerable. This means that completeness fails over finite models.

The technique of the proof, which is based on encoding Turing machine computations as finite structures, was reused by **Fagin** some 25 years later to prove his result putting an equality sign between the complexity class NP and existential second-order logic, hence providing a machine-independent characterization of an important complexity class.

As an example we may look at SQL (Structured Query Language), which is a well known - and one of the first - language for the relational database model described in Codd's 1970 paper. SQL is based on first-order predicate logic, and has the same expressive power.

Contents

1 **Introduction** 1
1.1 Model Theory 1
1.2 Finite Model Theory 1
1.3 Applications of Finite Model Theory 1

2 **Prerequisites** 3
2.1 Background from Mathematical Logic 3
2.2 Automata Theory 4
2.3 Computability Theory 7

3 **Second-Order Logic** 10

4 **Complexity Theory** 12
4.1 The Complexity Classes P and NP 12
4.2 Encodings of formulae and structures 13

5 **Trakhtenbrot's Theorem** 15
5.1 Trakhtenbrot's Theorem 15
5.2 Corollaries 15
5.3 Proof of Trakhtenbrot's Theorem 16

6 **Fagin's Theorem** 20
6.1 Fagin's Theorem 20
6.2 Proof of Fagin's Theorem 20

References 26

1 Introduction

1.1 Model Theory

Model theory is the study of models of theories in a formal language from the perspective of mathematical logic. With *theory* we mean a set of sentences (in a formal language), and a *model* of a theory is a structure (e.g. an interpretation) satisfying the sentences of that theory. Typically, this formal language is first order-logic (FOL) or some extension of FOL. More on the subject of FOL can be found in [1, ch. 2].

1.2 Finite Model Theory

Finite model theory (FMT) is a sub-area of model theory (MT), restricted to the study of finite structures (which, by definition, have a finite universe). Readers familiar with model theory will likely discover that some of the central (and often used) theorems of MT fail for finite structures, making FMT different from MT with regards to methods of proof.

1.3 Applications of Finite Model Theory

One of the most prominent field of application of FMT is in computer science, since structures of interest can be regarded as finite models.

SQL, Structured Query Language, is based on relational algebra, which in turn is based on first-order logic. Codd's theorem[1] states that relational algebra and the domain-independent relational calculus[2] queries are equivalent in expressive power. For example, assume we have a database table `students` with columns `first_name` and `last_name`, and assume all names are unique. This corresponds to a binary relation, say $G(f, l)$ on `first_name` $\times$ `last_name`.

[1] In the form of the stronger Equivalence Theorem in [2, ch. 5.3]

[2] Which is essentially equivalent to FO.

For example, the FO query $\{l : G('Max', l)\}$, returning all last names where the first name is 'Max', is expressed in SQL as:

```
SELECT last_name
FROM students
WHERE first_name = 'Max'
```

Furthermore, the programming paradigm of logic programming is based on formal logic. One of the major logic programming language families is Prolog, which is based on FOL. In Prolog, program logic is expressed in terms of relations, and computations are initiated by running queries over these relations.

Finite model theory has a strong connection to computability theory in general, and complexity theory in particular. In complexity theory, we classify computational problems according to their inherent difficulty, and relate those classes to each other. Two of the most fundamental complexity classes are P and NP. As a rule of thumb, one can say that P are the set of decision problems[3] which have "practical" solutions. NP contains all problems in P as well as some problems which probably has no practical solution[4]. However, the question whether P=NP or not is still open. More on the subject of P vs. NP can be found in e.g. [3, ch. 7.3].

When given some finite graph we may want to know whether it is Hamiltonian, which intuitively is the property of being a graph where one may find a circuit where all nodes are visited exactly once. The problem of testing whether a graph is Hamiltonian turns out to be an NP-complete problem, which means that the time needed for the best of all known algorithm for a graph of n vertices can not be bounded by a polynomial over n.

[3] Answerable with either yes or no.
[4] Called NP-complete problems.

2 Prerequisites

2.1 Background from Mathematical Logic

The following definitions are as defined in [4, pp. 13-16], with some minor alterations where needed.

Definition 1. A *vocabulary* σ is a collection of *constant* symbols (denoted $c_1, \ldots, c_n, \ldots$), *relation* (or *predicate*) symbols $(R_1, \ldots, R_n, \ldots)$ and *function* symbols $(f_1, \ldots, f_n, \ldots)$. Each relation and function symbol has an associated arity, i.e. the dimension of the domain.

A σ-structure, (also called a *model*)

$$\mathfrak{A} = \langle A, \{c_i^{\mathfrak{A}}\}, \{P_i^{\mathfrak{A}}\}, \{f_i^{\mathfrak{A}}\}\rangle$$

consists of a universe A together with an interpretation of:

- each constant symbol c_i from σ as an element $c_i^{\mathfrak{A}} \in A$;
- each k-ary relation symbol R_i from σ as a k-ary relation on A (a set $R_i^{\mathfrak{A}} \subseteq A^k$; and
- each k-ary function symbol f_i from σ as a k-ary function $f_i^{\mathfrak{A}} : a^k \to A$.

A structure $\mathfrak{A}$ is called *finite* if its universe A is a finite set.

Definition 2. A *theory* is a set of sentences. A structure $\mathfrak{A}$ is a model of a theory T iff for every sentence φ of T, $\mathfrak{A}$ is a model of φ; this is denoted $\mathfrak{A} \models \varphi$. A theory is called *consistent* if it has a model.

Theorem 3 (Completeness Theorem). *For a theory T and sentence φ, by $T \models \varphi$ we mean that when each sentence of T is true, so is φ. By $T \vdash \varphi$ we mean that φ is deducible from T in a formal proof system.*

The completeness theorem states that $T \models \varphi$ iff $T \vdash \varphi$.

Theorem 4 (Compactness Theorem). *A theory T is consistent iff every finite subset of T is consistent.*

Theorem 5 (Löwenheim-Skolem Theorem)**.** *If a countable theory T has an infinite model, it follows that T has a countable model.*

The proofs for the fundamental theorems 3, 4 and 5 are omitted. Interested readers can find proofs in Hedman [1, p. 167]. Later on, in section 5, we will see that the completeness- and Löwenheim-Skolem theorems fail in the finite case.

2.2 Automata Theory

Let Σ be a finite non-empty alphabet, i.e. a finite non-empty set of symbols. The set of all possible finite strings using characters from Σ is denoted Σ^*. Concatenation of two strings s and s' is denoted $s \cdot s'$ (or sometimes just ss', to be more concise). The empty string[5] is denoted ε. A *language* is a subset of Σ^*.

Non-deterministic versus Deterministic Automata

Definition 6. A *non-deterministic finite automaton* (*NFA* for short) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ where:

Q is the finite set of states;

Σ is a finite alphabet;

$q_0 \in Q$ is the initial state;

$F \subseteq Q$ is the set of final states, and;

δ is a transition function: $\delta : Q \times \Sigma \to P(Q)$, where $P(Q)$ denotes the power set of Q.

If $|\delta(q, a)| = 1$ for all $(q, a) \in Q \times \Sigma$ the automaton is called *deterministic* (DFA for short). Note that automatons are not partitioned into deterministic and non-deterministic automatons; the set of deterministic automatons are a subset of non-deterministic automatons.

[5]The unique string of length 0, i.e. without any symbols. For any string s, ε has the property $s \cdot \varepsilon = \varepsilon \cdot s = s$.

Definition 7. Let $s = a_1a_2a_3 \cdots a_n$ be a string in Σ^*. A *run* of $\mathcal{A}$ on s is defined as a mapping $r : \{1, \ldots, n\} \to Q$ such that

- $r(1) \in \delta(q_0, a_1)$ (if $\mathcal{A}$ is deterministic then $r(1) = \delta(q_0, a_1)$), and
- $r(i+1) \in \delta(r(i), a_{i+1})$ (if $\mathcal{A}$ is deterministic then $r(i+1) = \delta(r(i), a_{i+1})$)

If $r(n) \in F$ we say that the run is *accepting*, and that $\mathcal{A}$ *accepts* the string s if there is an accepting run. For a deterministic automaton there is exactly one run for each string while a non-deterministic automaton may have more than one run for a string. The set of strings accepted by $\mathcal{A}$ is denoted $L(\mathcal{A})$ and is called the *language of* $\mathcal{A}$.

Definition 8. A language L is called *regular* if there is a non-deterministic finite automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$. One can prove that for every regular language L there exists some deterministic finite automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$.

Turing Machines

Definition 9. A *Turing Machine* M is a tuple $(Q, \Sigma, \Delta, \delta, q_0, Q_a, Q_r)$ where:

Q is the finite (non-empty) set of states;

Σ is a finite *input* alphabet;

Δ is a finite tape alphabet containing Σ as well as a blank symbol '$\#$';

δ is a transition function: $\delta : Q \times \Delta \to 2^{Q \times \Delta \times \{\ell, r\}}$ (where ℓ, r stands for 'left' and 'right', respectively);

$q_0 \in Q$ is the initial state;

Q_a, Q_r are the sets of accepting and rejecting states respectively. Note that we require that $Q_a \cap Q_r = \emptyset$. We refer to states in $Q_a \cup Q_r$ as the *halting* states.

If the machine M will end in an accepting state when running on input w, we say that M *accepts* s. The set $\{s | M \text{ accepts } s\}$ is called the *language of* M, denoted $L(M)$.

In State	Reads Symbol	New State	New Symbol	Move
q_0	0	q_r	#	r
q_0	1	q_0	#	r
q_0	#	q_a	#	r

Table 1: Example of a transition function

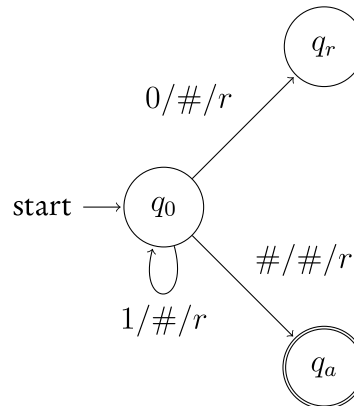

Figure 1: Diagram depicting the machine with transition function from table 1

Analogously to automatons, a Turing machine with $|\delta(q, a)| = 1$ for all $(q, a) \in Q \times \Sigma$ is called *deterministic* (or DTM for short). A *non-deterministic* Turing machine (NTM for short) may have $|\delta(q, a)| > 1$. Note that, analogously to automata, NTMs include DTMs as special cases.

Furthermore note that a defining characteristic for non-deterministic Turing machines is the ability to "guess". One can regard NTMs as branching "computational trees", whereas DTMs are non-branching "computational paths". If at least one of the branches of an NTM halts in an accepting state, we say that the NTM accepts the input. Therefore one can see a NTM as a "lucky guesser", always correctly guessing which branch to choose to get to an accepting state.

In table 1 and fig. 1, we see the transition function of a simple deterministic Turing machine M. Whenever it reads a zero, it goes to the rejecting state q_r. When M finds a blank symbol it accepts the input if and only if it is still in it's initial state. We conclude that M reads an input string of zeroes and ones and checks if the input contains no zeroes.

2.3 Computability Theory

Definition 10 (Recursively enumerable set)**.** A subset L of Σ^* is called *recursively enumerable* (or r.e. for short) if there is a Turing machine M such that $L = L(M)$.

Note that there are three outcomes when a Turing machine M runs on a string: M can halt in an accepting state, M can halt in a rejecting state, or M can go into an infinite loop and never halt. We call a Turing machine *halting* if the last outcome is impossible, in effect if M eventually enters a halting state on any input string s.

Definition 11 (Recursive set)**.** We call a subset L of Σ^* *recursive* if there is a halting Turing machine M such that $L = L(M)$.

We can regard halting Turing machines as deciders for some sets L: given some string s, M either enters an accepting or rejecting state when running on s, which decides whether or not $s \in L$. Therefore, one sometimes uses the term *decidable* instead of *recursive*. One then means that some encoding of the problem as a subset of Σ^* for some finite Σ is decidable.

Proposition 12. *A set A is recursive iff both A and A^c are r.e.*

Proof. Recursive sets are r.e.[6], and complements of recursive sets are recursive. This is because we can just redefine the halting Turing machine so that the rejecting states are accepting and vice versa in order to decide the complement.

For the converse, assume A and A^c are both r.e. Then there are two Turing machines[7] M_A and M_{A^c} where $A = L(M_A)$ and $A^c = L(M_{A^c})$. Now we can define a new Turing machine $\hat{M}$ where the new set of states is the union of the states of the two machines, where the two initial states are contracted into one. The new transition function contains all transitions of the two machines. The set of accepting states of $\hat{M}$ is the union of the set of accepting states of M_A and the rejecting states of M_{A^c}, and the set of rejecting states is constructed similarly.

[6] By definition 10.

[7] Not necessarily halting.

What we end up with is, intuitively, a machine which is the parallel composition of M_A and M_{A^c}. Any output from M_{A^c} is negated. For any string s we get that if $s \in A$, the M_A part will accept s. If $s \notin A$, we get that M_{A^c} will accept. Since $\hat{M}$ negates the output of M_{A^c}, $\hat{M}$ will reject s. This concludes the proof. $\square$

The Halting Problem

Definition 13 (Halting Set)**.** The *halting set* $\mathcal{H}$ is the set of all pairs $\langle M, s\rangle$ such that M is the encoding of a Turing machine accepting the string s.

The *halting problem* is the problem of determining whether some $\langle M, s\rangle$ is in $\mathcal{H}$. The undecidability of $\mathcal{H}$ is crucial to the proof of Trakhtenbrot's theorem in section 5.3.

Theorem 14 (The Halting Set is not Recursive)**.** *The problem of deciding whether or not a given $\langle M, s\rangle \in \mathcal{H}$ is undecidable.*

Proof. For a contradiction, assume there is some Turing machine H solving the Halting Problem. On input $\langle M, s\rangle$, H halts and accepts if the Turing machine M accepts s. Additionally, H halts and rejects if M fails to accept s. In other words, H has the following properties:

$$H(\langle M, s\rangle) = \begin{cases} \text{yes} & \text{if } H \text{ ends in } q_{\text{yes}} \\ \text{no} & \text{if } H \text{ ends in } q_{\text{no}} \end{cases}$$

Now construct the Turing Machine H' from H, calling H to determine what M does when the input to M is its own encoding $\langle M\rangle$. Once H' has determined, it does the opposite. In effect:

$$H'(\langle M\rangle) = \begin{cases} \text{yes} & \text{if } M \text{ does not accept } \langle M\rangle \\ \text{no} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

Now, for the contradiction, run $H'(\langle H'\rangle)$:

$$H'(\langle H'\rangle) = \begin{cases} \text{yes} & \text{if } H' \text{ does not accept } \langle H'\rangle \\ \text{no} & \text{if } H' \text{ accepts } \langle H'\rangle \end{cases}$$

No matter what H' does, it is forced to do the opposite, which is a contradiction. Hence no such H' can exist, which implies no such H can exist. This concludes the proof. □

3 Second-Order Logic

The idea of second-order logic is that, in addition to quantification over the elements of the universe, we are able to quantify over subsets over the universe, as well as relations on it.

Formally, we define it as follows:

Definition 15 (Formulae of second-order logic)**.** A formula of SO can have both first- and second-order free variables and we write $\varphi(\vec{x}, \vec{X})$ to indicate that $\vec{x}$ are free first-order variables and $\vec{X}$ are free second-order variables.

Given a vocabulary σ that consists of constant and relation symbols, we define SO terms and formulae, and their free variables, as follows:

- Every first-order variable x, and every constant symbol c, are first-order terms. The only free variable of a term x is the variable x, and the constant c has no free variables.
- There are three kinds of atomic formulae, namely of one of the following forms:
  - $t = t'$, where t, t' are terms;
  - $R(\vec{t})$, where $\vec{t}$ is a n-tuple of terms, and R is a n-ary relation symbol in σ; and
  - $X(\vec{t})$, where $\vec{t}$ is a n-tuple of terms, and X is a second-order variable of arity n. The free first-order variables of this formula are free first-order variables of $\vec{t}$; the free second-order variable is X.
- SO-formulae are closed under the Boolean connectives $\vee, \wedge, \neg$, and first order quantification, with the usual rules for free variables.
- If $\varphi(\vec{x}, \vec{X}, Y)$ is a formula, then so are $\exists Y \varphi(\vec{x}, \vec{X}, Y)$ and $\forall Y \varphi(\vec{x}, \vec{X}, Y)$.

Most of the semantics are inherited from FO, but we need to define some new semantics:

Definition 16 (Semantics of second-order logic)**.** Suppose $\mathfrak{A}$ is a σ-structure. For each formula $\varphi(\vec{x}, \vec{X})$, we define the notion $\mathfrak{A} \models \varphi(\vec{b}, \vec{B})$, where $\vec{b}$ is a tuple of elements of A of the same length as $\vec{x}$, and for $\vec{X} = (X_1, \dots, X_\ell)$ with each X_i being n_i-ary, $\vec{B} = (B_1, \dots, B_\ell)$, where each $B_i \subseteq A^{n_i}$.

- If $\varphi(\vec{x}, X)$ is $X(t_1, \dots, t_k)$, where X is k-ary and t_i's are terms, with free variables among $\vec{x}$, then, $\mathfrak{A} \models \varphi(\vec{b}, B)$ iff the tuple $t_1^{\mathfrak{A}}(\vec{b}), \dots, t_k^{\mathfrak{A}}(\vec{b})$ is in B.
- If $\varphi(\vec{x}, \vec{X})$ is $\exists Y \psi(\vec{x}, \vec{X}, Y)$, where Y is k-ary, then $\models \varphi(\vec{b}, \vec{B})$ if *there is some* $C \subseteq A^k$ such that $\mathfrak{A} \models \psi(\vec{b}, \vec{B}, C)$.
- If $\varphi(\vec{x}, \vec{X})$ is $\forall Y \psi(\vec{x}, \vec{X}, Y)$, where Y is k-ary, then $\models \varphi(\vec{b}, \vec{B})$ if *for all* $C \subseteq A^k$ we have $\mathfrak{A} \models \psi(\vec{b}, \vec{B}, C)$.

Definition 17 (Existencial SO logic, abbr. $\exists$SO)**.** A SO formula is in $\exists$SO iff it can be written on the form

$$\exists X_1 \dots \exists X_n \varphi$$

where φ is second-order-quantifier free.

In other words, an $\exists$SO formula can be written as such that it starts with a second-order existential prefix and ends with an FO formula. In section 6 we will see a convenient result regarding the expressibility of $\exists$SO.

4 Complexity Theory

4.1 The Complexity Classes P and NP

Let L be a language accepted by a halting Turing machine M. Assume that there is some function $f : \mathbb{N} \to \mathbb{N}$ such that the number of transitions of states M makes before accepting or rejecting a string s is bounded from above by $f(|s|)$ (where $|s|$ is the length of s). If M is deterministic, we write $L \in DTIME(f)$ and if M is non-deterministic we write $L \in NTIME(f)$.

We now define the class P of polynomial-time computable problems as

$$P := \bigcup_{k \in \mathbb{N}} DTIME(n^k)$$

and the class NP of problems computable by non-deterministic polynomial-time Turing machines as

$$NP := \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

Intuitively, P can be seen as the problems that are relatively easy both to solve and check a solution. NP may be hard to solve, but it should be easy to check solutions; The lucky-guessing NTM guesses the solution on it's first try when solving.

Since the set of deterministic Turing machines is a subset of the non-deterministic ones, it follows that

$$P \subseteq NP$$

It is now known whether the inclusion is proper and the "P versus NP problem" is one of the most prominent unsolved problems in computer science, asking whether $P = NP$. This would be quite remarkable since it would mean problems eluding computer scientists for decades actually have a simpler solution.

For example public-key cryptography, which usually relies on the prime

factorization problem being difficult[8] to solve. To show that $P = NP$ would imply that there is some Turing machine solving the problem from NP in polynomial time, leading to factorization probably not being hard enough for safe use for cryptographical purposes.

4.2 Encodings of formulae and structures

Complexity theory defines its main concepts via acceptance of string languages by computational devices, such as Turing machines. Therefore, to talk about complexity of logics on finite structures, we need to encode finite structures and logical formulae as strings. For formulae, we shall assume some natural encoding: for example, enc(φ) – the encoding of the formula φ, could be its syntactic tree represented as a string. For the notion of data complexity, the choice of a particular encoding of formulae does not matter.

$$(P \lor Q) \land (\neg T \to S)$$

$$P \lor Q \qquad \neg T \to S$$

$$P \quad Q \qquad \neg T \quad S$$

$$T$$

Figure 2: Syntactic tree of a formula.

When encoding structures, there are several different ways to do so. We will use the most often used. Suppose we have a σ-structure $\mathfrak{A}$. Let $A = \{a_1, \ldots, a_n\}$. For encoding a structure, we always assume an ordering on the universe. In some structures, the order relation is part of the vocabulary, but in others it is not. In the latter case, we may arbitrarily choose one; the order in this case will have no effect on the result of queries, but we need it to represent the encoding of a structure on a Turing machine's tape, to be able to talk about computability and complexity of queries.

[8] In the sense of it taking non-polynomial time to execute an algorithm solving the problem.

Thus, we choose an order on the universe: for simplicity, let us choose $a_1 < a_2 < \ldots < a_n$. Each k-ary relation $R^{\mathfrak{A}}$ will be encoded by an n^k-bit string $\text{enc}(R^{\mathfrak{A}})$ as follows: Consider an enumeration of *all* k-tuples over A, in the lexicographic order:

$$(a_1, \ldots, a_1), (a_1 \ldots, a_1, a_2), \ldots, (a_n, \ldots, a_n, a_{n-1}), (a_n, \ldots, a_n)$$

Let $\vec{a}_j$ be the jth tuple in this enumeration. Then the jth bit of $\text{enc}(R^{\mathfrak{A}})$ is 1 if $\vec{a}_j \in R^{\mathfrak{A}}$, and 0 if $\vec{a}_j \notin R^{\mathfrak{A}}$. We shall assume, without loss of generality, that σ contains only relation symbols, since we can encode a constant as a unary relation containing one element.

If $\sigma = \{R_1, \ldots, R_p\}$, then the basic encoding of a structure is the concatenation of the encodings of relations: $\text{enc}(R_1^{\mathfrak{A}}) \cdots \text{enc}(R_p^{\mathfrak{A}})$. In some computational models, the length of the input is a parameter of the model and thus $|A|$ can easily be calculated from the basic encoding. In others, e.g. Turing machines, $|A|$ must be known by the device in order to use the encoding of a structure. For that purpose, we define an $\text{enc}(\mathfrak{A})$ which simply is the concatenation of the string 0^n1 and all of the $\text{enc}(R_i^{\mathfrak{A}})$'s:

$$\text{enc}(\mathfrak{A}) = 0^n1 \cdot \text{enc}(R_1^{\mathfrak{A}}) \cdots \text{enc}(R_p^{\mathfrak{A}})$$

The length of this string, denoted by $||\mathfrak{A}||$, is

$$||\mathfrak{A}|| = (n+1) + \sum_{i=1}^{p} n^{\text{arity}(R_i)}$$

5 Trakhtenbrot's Theorem

5.1 Trakhtenbrot's Theorem

Definition 18 (Finite satisfiability, finite validity)**.** Given a vocabulary σ, a sentence φ in that vocabulary is called *finitely satisfiable* if there is a finite σ-structure $\mathfrak{A}$ such that $\mathfrak{A} \models \varphi$.

The sentence φ is called *finitely valid* if $\mathfrak{A} \models \varphi$ holds for all finite σ-structures $\mathfrak{A}$.

Theorem 19 (Trakhtenbrot's Theorem)**.** *For every relational vocabulary σ with at least one binary relation symbol, it is undecidable whether a first-order sentence φ of vocabulary σ is finitely satisfiable.*

Before proving the theorem, I will state and prove a couple of important corollaries.

5.2 Corollaries

Recall definition 10: a subset L of Σ^* is called recursively enumerable if there is a Turing machine M such that L is the language of M. In other words, L is *exactly* the set of strings that make M end in an accepting state.

Corollary 20. *For any vocabulary containing at least one binary relation symbol, the set of finitely valid sentences is not recursively enumerable.*

Note that corollary 20 implies the failure of the analogue to completeness theorem in the finite case. Recall that the completeness theorem for FO states that a sentence φ is true in all models iff it is provable in some formal proof system. Since we can enumerate all formal proofs of valid FO sentences, the set of all valid FO sentences is recursively enumerable.

Proof. For a contradiction, assume that the set of finitely valid sentences is recursively enumerable. Given a sentence φ, we can consider each of the finitely many structures up to isomorphism in the given vocabulary having size n for

$n = 1, 2, 3, \ldots$ If φ has a finite model, then we would find such a model in a finite number of steps. Hence, the set of finitely satisfiable sentences is r.e.

By theorem 19, we know that the set of finitely satisfiable sentences is not recursive, so the complement of this set cannot be recursively enumerable (because then the set would be recursive by proposition 12). The complement is the set of all sentences that are not satisfiable in a finite structure. A sentence φ is not finitely satisfiable iff $\neg\varphi$ is finitely valid. It follows that the set of finitely valid sentences is not recursively enumerable. □

Corollary 21. *There is no recursive function f such that if φ has a finite model, then it has a model of size at most $f(\varphi)$.*

Note that corollary 21 implies the failure of the analogue to the Löwenheim-Skolem Theorem for finite models.

Proof. If there was such a function calculating an upper bound of model size one would certainly be able to decide finite satisfiability by testing all models up to that size. Thus, this would be in direct contradiction to Trakhtenbrot's theorem. □

5.3 Proof of Trakhtenbrot's Theorem

The proof, as presented in [4], is based on the idea that we, given a Turing machine M construct a sentence φ_M of vocabulary σ such that φ_M is finitely satisfiable if and only if M halts on the empty input. By this, we reduce the problem to The Halting Problem on the empty input which is undecidable by theorem 14. If we can define such a φ_M we may deduce that the problem of finite satisfiability, too, is undecidable.

Proof. Let

$$M_e = (Q, \Sigma, \Delta, \delta, q_0, Q_a, Q_r)$$

be a deterministic Turing machine with a one-way infinite tape. Q is the set of states, Σ the input alphabet, Δ is the tape alphabet (including the blank symbol), q_0 the initial state, Q_a and Q_r is the set of accepting states and the set of rejecting states respectively (from which there are *no* transitions), and finally

δ is the transition function. Since we are coding the problem of halting on the empty input, we may assume without loss of generality that $\Delta = \{0, 1\}$ with 0 playing the role of the blank symbol.

Define σ so that its structures represent computations of M as such:

$$\sigma = \{<, \underline{\min}, T_0(\cdot,\cdot), T_1(\cdot,\cdot)\} \cup \{(H_q(\cdot,\cdot)) : q \in Q\}$$

where

- $<$ is a linear order and $\underline{\min}$ is a constant symbol for the minimal element with respect to $<$. In other words, the finite universe will be associated with an initial segment of the natural numbers starting from $\underline{\min}$.
- T_0 and T_1 are tape predicates; $T_i(p, t)$ means that position p at time t contains i (for $i \in \Delta$).
- H_q's are head predicates; $H_q(p, t)$ means that at time t, the machine is in state q, and its head is in position p.

We want the sentence φ_M to state that $<$, $\underline{\min}$, T_i's and H_q's are interpreted as indicated above and that the machine eventually halts. Note that if the machine halts, then $H_q(p, t)$ holds for some p, t and $q \in Q_a \cup Q_r$, and that after that the configuration of the machine does not change. That is, all the configurations of the halting computation can be represented by a finite σ-structure.

We define φ_M to be the conjunction of the following sentences:

- A sentence stating that $<$ is a linear order and $\underline{\min}$ is its minimal element.
- A sentence defining the starting configuration of M:

  $$H_{q_0}(\underline{\min}, \underline{\min}) \wedge \forall p T_0(p, \underline{\min})$$

  which states that M is in state q_0, the head is in the first position and the tape is blank (it contains only zeros).

- A sentence stating that, in every configuration of M, each cell of the tape contains either 0 or 1, but not both:

$$\forall p \forall t (T_0(p,t) \leftrightarrow \neg T_1(p,t))$$

- A sentence stating that the machine, at any time, is in exactly one state:

$$\forall t \exists! p \left(\bigvee_{q \in Q} H_q(p,t) \right) \wedge \neg \exists p \exists t \left(\bigvee_{q,q' \in Q, q \neq q'} H_q(p,t) \wedge H_{q'}(p,t) \right)$$

- Furthermore, we need a set of sentences stating that the T_i's and H_q's respect the transitions of M, with one sentence per transition.

  For example, assume that if M is in state q reading 0, it writes 1, moves the head one position to the left and changes states to q'. Using our mathematical notation, we write this:

$$\delta(q,0) = (q', 1, \ell)$$

  and this transition is represented by the conjunction of the two sentences:

$$\forall p \forall t \begin{pmatrix} & p \neq \underline{\min} \\ \wedge & T_0(p,t) \\ \wedge & H_q(p,t) \end{pmatrix} \rightarrow \begin{pmatrix} & & T_1(p,t+1) \\ \wedge & & H_{q'}(p-1,t+1) \\ \wedge & \forall p' & \begin{pmatrix} p \neq p' \rightarrow \\ \bigwedge_{i \in \{0,1\}} T_i(p',t+1) \leftrightarrow T_i(p',t) \end{pmatrix} \end{pmatrix}$$

  and

$$\forall p \forall t \begin{pmatrix} & p = \underline{\min} \\ \wedge & T_0(p,t) \\ \wedge & H_q(p,t) \end{pmatrix} \rightarrow \begin{pmatrix} & & T_1(p,t+1) \\ \wedge & & H_{q'}(p,t+1) \\ \wedge & \forall p' & \begin{pmatrix} p \neq p' \rightarrow \\ \bigwedge_{i \in \{0,1\}} T_i(p',t+1) \leftrightarrow T_i(p',t) \end{pmatrix} \end{pmatrix}$$

Here "$p-1$" and "$t+1$" are short-hand for "the greatest element less than p" and "the smallest element greater than t", respectively. The difference between the two sentences is simply that $p-1$ when $p = \underline{\min}$ is undefined, so we let the machine stay if it is already in the first position and tries to go left.

- And finally, since we want M to be halting, we need a sentence stating that M is in a halting state at some point:

$$\exists p \exists t \bigvee_{q \in Q_a \cup Q_r} H_q(p,t)$$

If φ_M indeed has a finite model, then such a model represents a computation of M that starts with the tape containing all zeros (i.e. the empty input), and ends in a halting state. Conversely, if M halts on the empty input, then the set of all configurations of the halting computation of M coded as relations $<$, T_i's, and H_q's, is a model of φ_M (which necessarily is finite). Thus, M halts on the empty input *if and only if* φ_M has a finite model. By undecidability of halting on the empty string (by theorem 14), finite satisfiability for φ_M is undecidable. □

6 Fagin's Theorem

Definition 22. Let $\mathcal{K}$ be a complexity class, $\mathcal{L}$ a logic and $\mathcal{C}$ a class of finite structures. We say that $\mathcal{L}$ *captures* $\mathcal{K}$ on $\mathcal{C}$ if the following hold:

1. The data complexity of $\mathcal{L}$ on $\mathcal{C}$ is $\mathcal{K}$; that is, for every $\mathcal{L}$-sentence φ, testing if $\mathfrak{A} \models \varphi$ is in $\mathcal{K}$, provided $\mathfrak{A} \in \mathcal{C}$.

2. For every property $\mathcal{P}$ of structures from $\mathcal{C}$ that can be tested with complexity $\mathcal{K}$, there is a sentence $\varphi_{\mathcal{P}}$ of $\mathcal{L}$ such that $\mathfrak{A} \models \varphi_{\mathcal{P}}$ if and only if $\mathfrak{A}$ has the property $\mathcal{P}$, for every $\mathfrak{A} \in \mathcal{C}$

If $\mathcal{C}$ is the class of all finite structures, we simply say that $\mathcal{L}$ *captures* $\mathcal{K}$.

6.1 Fagin's Theorem

Theorem 23 (Fagin's Theorem). *$\exists SO$ captures NP.*

Although very quickly stated, Fagin's theorem[9] is a very significant result as it was the first machine-independent characterization of a complexity class. Usually one would need to refer to some kind of computational model such as a Turing machine.

6.2 Proof of Fagin's Theorem

Proof. First, we show that every $\exists SO$-sentence ϕ can be evaluated in NP. We remind ourselves that a characteristic of non-deterministic Turing machines is the ability to "guess". Suppose ϕ is $\exists S_1 \dots \exists S_n \varphi$ where φ has only first-order quantifiers. Given $\mathfrak{A}$, the non-deterministic machine guesses $S_1, \dots, S_n$ and checks if $\varphi(S_1, \dots, S_n)$ holds. The latter can be done in polynomial time in $||\mathfrak{A}||$ plus the size of $S_1, \dots, S_n$, hence polynomial time in $||\mathfrak{A}||$.

[9] First presented in [6].

Second, we show that every NP property of finite structures can be expressed in $\exists$SO. Libkin's [4] proof of this direction is similar to the proof of Trakhtenbrot's theorem, but we now need to consider two additional elements: namely time bounds, and the input.

Suppose we are given a property $\mathcal{P}$ of σ-structures that can be tested on encodings of σ-structures, by a non-deterministic polynomial time Turing machine $M = (Q, \Sigma, \Delta, \delta, q_0, Q_a, Q_r)$ with a one-way infinite tape. Here, $Q = \{q_0, \dots, q_{m-1}\}$ is the set of states, and we may assume, without loss of generality, that $\Sigma = \{0, 1\}$ and that Δ extends Σ with a blank symbol "#". Furthermore, we assume that there is some k such that M runs in time n^k. Moreover, we assume that k is greater than the arity of the relations in σ. Note that n is the size of the encoding, so we must assume $n > 1$. We may also assume, again without loss of generality, that M always visits the entire input; in effect, n^k always exceeds the size of the encodings of n-element structures.

The formula stating the fact that M accepts an encoding of a σ-structure will assume the form

$$\exists L \exists T_0 \exists T_1 \exists T_2 \exists H_{q_0} \dots \exists H_{q_{m-1}} \Psi \tag{1}$$

where Ψ is a first-order sentence of vocabulary $\sigma \cup \{L, T_0, T_1, T_2\} \cup \{H_q | q \in Q\}$. Here L is binary, and other symbols are of arity $2k$. The intended interpretation of these relational symbols is as follows:

- L is a linear order on the universe.

Using L, one can now define, in FO, the lexicographic linear order $\leq_k$ on k-tuples. Since M runs in time n^k and visits at most n^k cells, we can model both positions on the tape ($\vec{p}$) and time ($\vec{t}$) by k-tuples of the elements of the universe.

For example, in the case $n = 2, k = 5$ step number 1 could be coded as $(0,0,0,0,0)$ and step number 28 could be coded as $(1,1,0,1,1)$, i.e. the binary encodings of the numbers 0 and 27. Since we know by assumption that we need at most $n^k = 2^5 = 32$ time steps, we also know that we need at most 32 different elements to differentiate between any two steps in time. A suitable choice could be the 5-character string corresponding to the binary representation of the counting numbers $\{0, \dots, 31\}$, however we could choose another system.

The predicates T_i's and H_q's are to be interpreted similarly to the proof of Trakhtenbrot's theorem:

- T_0, T_1 and T_2 are *tape* predicates; $T_i(\vec{p}, \vec{t})$ means that position $\vec{p}$ at time $\vec{t}$ contains i, for $i = 0, 1$, and $T_2(\vec{p}, \vec{t})$ says that $\vec{p}$ at time $\vec{t}$ contains the blank symbol.
- The H_q's are *head* predicates; $H_q(\vec{p}, \vec{t})$ means that at time $\vec{t}$ the machine is in state q, and its head is in position $\vec{p}$.

The sentence Ψ must now assert that when M starts on the encoding of $\mathfrak{A}$, the predicates T_i's and H_q's correspond to M's computation, and that M can reach an accepting state. Note that the encoding of $\mathfrak{A}$ depends on a linear ordering on the universe of A. We may assume, without loss of generality, that this ordering is L.

We now define Ψ as the conjunction of the following sentences:

- The sentence stating that L defines a linear ordering.
- The sentence stating that:
  - in every configuration of M, each cell of the tape contains exactly one element of Δ;
  - at any time the machine is in exactly one state;
  - eventually, M enters a state from Q_a.

  All of these sentences are expressed in the exact same way as in the proof of Trakhtenbrot's theorem (starting on page 16).

- Sentences stating that the T_i's and H_q's respect the transitions of M, which is done very similarly as in the proof of Trakhtenbrot's theorem. In the proof of Trakhtenbrot's theorem we had a deterministic TM, now we have to take non-determinism into account. For every $a \in \Delta$ and $q \in Q$, we have a sentence

$$\bigvee_{(q',b,move)\in\delta(q,a)} \alpha_{(q,a,q',b,move)}$$

where $move \in \{\ell, r\}$, and $\alpha_{(q,a,q',b,move)}$ is the sentence describing the transition where the machine reads a in state q, writes b, makes the move $move$, and enters state q'. Such a sentence is written in the exact same way as in the proof of Trakhtenbrot's theorem.

- The sentence defining the initial configuration of M. Suppose we have formulae $\iota(\vec{p})$ and $\zeta(\vec{p})$ of vocabulary $\sigma \cup \{L\}$ such that $\mathfrak{A} \models \iota(\vec{p})$ iff the $\vec{p}$th position of enc($\mathfrak{A}$) is 1 (in the encoding presented in section 4.2), and $\mathfrak{A} \models \zeta(\vec{p})$ iff $\vec{p}$ exceeds the length of enc($\mathfrak{A}$). Note that we need L in these formulae since the encoding refers to a linear order on the universe. With such formulae, we define the initial configuration by

$$\forall \vec{p} \forall \vec{t} \left(\neg \exists \vec{u} (\vec{u} <_k \vec{t}) \rightarrow \left[\begin{pmatrix} & (\iota(\vec{p}) \leftrightarrow T_1(\vec{t}, \vec{p})) \\ \wedge & (\zeta(\vec{p}) \leftrightarrow T_2(\vec{t}, \vec{p})) \end{pmatrix} \right] \right)$$

In effect, at time 0, the tape contains the encoding of the structure followed by blanks.

As in the proof of Trakhtenbrot's theorem, we conclude that eq. (1) holds in $\mathfrak{A}$ iff M accepts enc($\mathfrak{A}$). Hence, it remains to show how to define the formulae $\iota(\vec{p})$ and $\zeta(\vec{p})$.

In order to keep the notation a bit more manageable, we will illustrate this with the case $\sigma = \{E\}$, with E binary (hence viewable as a graph). Extension to arbitrary vocabularies is straightforward. Assume that the universe of the graph is $\{0, \ldots, n-1\}$, where $(i, j) \in L$ iff $i < j$. The graph is encoded by the string $0^n 1 \cdot s$, where s is a string of length n^2 such that is has 1 in position

$u \cdot n + v$ for $0 \leq u, v \leq n - 1$ iff $(u, v) \in E$. The actual encoding of E starts in position $(n + 1)$ since the first $n + 1$ positions are just for describing the size of the graph.[10]

$$\mathrm{enc}(E) = \overbrace{\underbrace{0 \ldots 0}_{n \text{ zeros}} 1}^{0 \text{ through } n} \quad \overbrace{\underbrace{\{0, 1\}^{n^2}}_{\text{encoding of } E}}^{n+1 \text{ through } n^2+n}$$

Here one can see that in the presence of addition and multiplication, ι is definable. $\vec{p} = (p_1, \ldots, p_k)$ represents the position $p_1 \cdot n^{k-1} + p_2 \cdot n^{k-2} + \ldots + p_{k-1} \cdot n + p_k$. Hence, $\iota(\vec{p})$ is equivalent to the disjunction of the following two formulae:

$$\sum_{i=1}^{k} p_i \cdot n^{k-i} = n \tag{2}$$

$$\exists u \leq (n-1) \exists v \leq (n-1) \left((n + 1) + u \cdot n + v = \sum_{i=1}^{k} p_i \cdot n^{k-i} \wedge E(u, v) \right) \tag{3}$$

Eq. 2 simply says that the number described by $\vec{p}$ is n, which by construction of the encoding is 1. Eq. 3 says that the number described by $\vec{p}$ is the $(u \cdot n + v)^{\text{th}}$ character of the encoding of E and that (u, v) is an edge in the graph. So the disjunction of the two sentences would, in English, say:

> "Either we are looking at character number n, or we are looking at character number $u \cdot n + v$ of the encoding of E and (u, v) is an edge."

With addition and multiplication this is definable, and addition and multiplication can be introduced by means of additional existential second-order quantifiers (since one can state in FO that a given relation properly represents addition or multiplication with respect to the ordering L).

Although this is enough to conclude definability of ι, we now sketch a proof of definability of ι without any additional arithmetic. Instead, we shall only refer to the linear ordering L, and we shall use the associated successor relation.

[10] Remember that we start counting from 0.

Assume $k = 3$. This means a tuple $\vec{p}$ represents the position $p_1 n^2 + p_2 n + p_3$ on the tape. The first position where the encoding of E starts is $(n + 1)$ since the positions 0 through n represent the size of the universe. The last position of the encoding is $n^2 + n$.

Hence, if $p_1 > 1$ then $\vec{p}$ represents a position $p \geq 2n^2 + p_2 n + p3$ which can not be on the tape. We conclude ι is *false* so $p_1 = 0$ or 1.

Assume $p_1 = 0$. Then we are talking about the position $p_2 n + p_3$. Remember $\iota(\vec{p})$ says that the position $\vec{p}$ contains 1. Positions 0 to $n - 1$ have zeros, so if $p_2 = 0$ then again ι is *false*.

If $p_3 \neq 0$, then $(p_2 - 1)n + (p_3 - 1) + (n + 1) = p_2 n + p_3$. Remember, $\text{enc}(E)$ is a string $0^n 1 \cdot s$ such that position $u \cdot n + v$ of s has a 1 iff $E(u, v)$. Position $p_2 n + p_3$ of $\text{enc}(E)$ corresponds to position $(p_2 - 1)n + (p_3 - 1)$ of s. Hence the position corresponds to $E(p_2 - 1, p_3 - 1)$.

If $p_3 = 0$, then this position corresponds to $E(p_2 - 2, n - 1)$. Hence, the formula $\iota(p_1, p_2, p_3)$ is of the form

$$
\begin{aligned}
&\left[\begin{pmatrix} & (p_1 = 0) \\ \wedge & (p_2 > 1) \end{pmatrix} \wedge \begin{pmatrix} & (p_3 \neq 0) \wedge E(p_2 - 1, p_3 - 1) \\ \vee & (p_3 = 0) \wedge E(p_2 - 2, n - 1) \end{pmatrix}\right] \\
&\vee \Big[(p_1 = 0) \wedge (p_2 = 1) \wedge (p_3 = 0)\Big] \vee \Big[(p_1 = 1) \wedge \ldots\Big]
\end{aligned}
$$

where in the case of $p_1 = 1$ a similar case analysis is made. Extension of the procedure for arbitrary values of k is straight-forward. Clearly, with the linear order L both 0 and $n - 1$, and the predecessor function are definable, and hence ι is FO. The formula $\zeta(\vec{p})$ simply says that the vector $\vec{p}$, considered as a number in the way described, exceeds $n^2 + n + 1$.

This completes the proof of Fagin's theorem.

□

References

[1] S. Hedman, *A First Course in Logic*.
Oxford University Press, 2004.

[2] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*.
Addison-Wesley, 1995.

[3] M. Sipser, *Introduction to the Theory of Computation*.
Thomson, 2006.

[4] L. Libkin, *Elements of Finite Model Theory*.
Springer, 2004.

[5] H.-D. Ebbinghaus, J. Flum, *Finite Model Theory*.
Springer, 1999.

[6] R. Fagin, *Generalized First-Order Spectra and Polynomial-Time Recognizable Sets*.
Complexity of Computation, ed. R. Karp, 1974.