

GYMNASIUM BÄUMLIHOF

MATURAARBEIT

Theoretical Informatics: Formal languages and finite model theory

A study of the connection of first order logic and
context-sensitive languages

Written by:
Yaël Arn, 4A



Platzhalter für Titelbild

Supervisor:
ALINE SPRUNGER

Coreferent:
BERNHARD PFAMMATTER

19th July 2024, 4058 Basel

Contents

1	Introduction	4
2	Formal Languages	5
2.1	Definition	5
2.2	Chomsky Hierarchy	5
2.2.1	Grammars	5
2.2.2	Regular Languages	6
2.2.3	Context-Free Languages	7
2.2.4	Context-Sensitive Languages	7
2.2.5	Recursive Languages	8
3	Descriptive Complexity	9
3.1	Aims	9
3.2	Tools	9
3.2.1	Complexity Theory	9
3.2.2	Reduction and Completeness	10
3.2.3	Ehrenfeucht-Fraïssé Games	11
3.3	Important Results	12
3.3.1	$\text{NSPACE}[s(n)] \subseteq \text{DSPACE}[s(n)^2]$	12
3.3.2	SPACE Hierarchy theorem	12
3.4	Results concerning the Chomsky hierarchy	13
3.4.1	Regular Languages	13
3.4.2	Context-Free Languages	13
3.4.3	Context-Sensitive Languages	13
3.4.4	Recursive Languages	13
3.5	Open questions	13
3.5.1	$\text{P} \stackrel{?}{=} \text{NP}$	13
3.5.2	$\text{NSPACE}[O(n)] \stackrel{?}{=} \text{DSPACE}[O(n)]$	13
4	Personal Contribution	14
5	Results	15
6	Conclusion and Direction	16
	Index	18
	List of Figures	18
	Listings	19

Bibliography	20
A Mathematical Background	21
A.1 Set Theory	21
A.2 First Order Logic	21
A.3 Second Order Logic	23
A.4 Turing Machines	23

Forword

1. Introduction

2. Formal Languages

2.1 Definition

In informatics, we often get an input as a string of characters, and want to compute some function on it. In complexity Theory, we mostly focus on decision problems where we only want to find out if some input fulfills some given property. To formalize this, there is the concept of formal languages. The following definitions are taken from the lecture Theory of Computer Science [Rög23]. For the mathematical background, refer to Appendix A.

Definition 2.1 (Alphabet). An alphabet Σ is a finite set of symbols

Definition 2.2 (Word). A word over some alphabet Σ is finite sequence of symbols from Σ . We denote ε as the empty word, Σ^* as the set of all words over Σ and $|w|$ as the number of symbols in w .

The concatenation of two words or symbol is written after each other, examples are ab and $\Sigma^*a\Sigma^*$ (the set of all words containing at least one a).

Definition 2.3 (Formal Language). A formal language is a set of words over some alphabet Σ , that is a subset of Σ^*

For any computational decision problem, we can then reformulate it as the problem of deciding if the input word is contained in the formal language consisting of all words which have the required property.

2.2 Chomsky Hierarchy

One of the multiple ways to categorize formal languages was invented by Avram Noam Chomsky, a modern linguist. It is based on the complexity of defining the language in some finite way, namely using grammars, but other formalisms are equivalent.

2.2.1 Grammars

A grammar can informally be seen as a set of rules telling us how to generate all words in a language.

Definition 2.4 (Grammar). A grammar is a 4-tuple $\langle V, \Sigma, R, S \rangle$ consisting of

V The set of non-terminal symbols

Σ The set of terminal symbols

R A set of rules, formally over $(V \cup \Sigma)^*V(V \cup \Sigma)^* \times (V \cup \Sigma)^*$

S The start symbol from the set V

The non-terminal symbols are symbols that are not in the end alphabet Σ and exist for the purpose of steering the process of word generation. Further, the rules dictate that there must be at least one non-terminal symbol on the left-hand side of the production rule, as $(V \cup \Sigma)^*$ contains all words consisting of symbols from V and Σ , and thus $(V \cup \Sigma)^*V(V \cup \Sigma)^*$ is the language of all words containing at least one non-terminal symbol. We normally write rules in the form $a \rightarrow b$ instead of $\langle a, b \rangle$.

To generate the words, we have the concept of derivations.

Definition 2.5 (Derivation). First, we can define one derivation step.

We say u' can be derived from u if

- u is of the form xyz for some words $x, y, z \in (V \cup \Sigma)^*$ and u' is of the form $xy'z$
- there exists a rule $y \rightarrow y'$ in R

We say that a word is in the *generated language* of a grammar if it can be derived in a finite number of steps from S .

Example 2.1. Consider the grammar $\langle \{S\}, \{a, b\}, R, S \rangle$ with

$$R = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$$

The generated language for this grammar is $\{\varepsilon, ab, aabb, \dots\} = \{a^n b^n \mid n \in \mathbb{N}_0\}$

Now that we have a tool to describe some infinite languages using a finite description, we can further differentiate the complexity of a language by the minimum required complexity of the rules in any grammar that describes the language.

2.2.2 Regular Languages

The regular languages have the most restricted type of grammars. Formally, any regular language can be described by a grammar with rules in $V \times (\Sigma \cup \Sigma V \cup \varepsilon)$. This means that we only have exactly one non-terminal on the left-hand side and the right hand side is either a terminal, the empty word or a terminal symbol followed by a non-terminal symbol.

Example 2.2. Consider the grammar $\langle \{S, O\}, \{a\}, R, S \rangle$ with

$$R = \left\{ \begin{array}{l} S \rightarrow aO, \quad S \rightarrow \varepsilon, \\ O \rightarrow aS \end{array} \right\}$$

The generated language are exactly all words with even length.

These languages have been studied quite thoroughly and have multiple equivalent formalisms:

- The language is recognised Deterministic finite automata, which process the input word one character at a time (for a formal definition see [Rög23])

- The language can be decided by a read-only turing machine, that is a turing machine that can not modify it's tape
- The language can be described by a regular expression

For a more in-depth analysis of regular languages and equivalent formalisms refer to section 3.4.1 and [Str94].

2.2.3 Context-Free Languages

The context-free languages extend the regular languages by allowing arbitrary right-hand sides for the rules of the defining grammar. Formally, that gives us rules in $V \times (\Sigma \cup V)^*$. Most valid arithmetic expressions, logical formulas and formally correct code in programming languages are context-free, as we can see the non-terminal symbols as types which are then converted to specific expressions of that type.

Example 2.3. Consider the grammar $\langle \{\mathbf{Exp}, \mathbf{NumF}, \mathbf{Num}\}, \{0, 1, (,), -, +\}, R, \mathbf{Exp} \rangle$ with

$$R = \left\{ \begin{array}{ll} \mathbf{Exp} \rightarrow \mathbf{NumF}, & \mathbf{Exp} \rightarrow (\mathbf{Exp} + \mathbf{Exp}), \\ \mathbf{Exp} \rightarrow (\mathbf{Exp} - \mathbf{Exp}), & \mathbf{Exp} \rightarrow (-\mathbf{Exp}), \\ \mathbf{Num} \rightarrow 0\mathbf{Num}, & \mathbf{Num} \rightarrow 1\mathbf{Num}, \\ \mathbf{Num} \rightarrow \varepsilon, & \mathbf{NumF} \rightarrow 0, \\ \mathbf{NumF} \rightarrow 1\mathbf{Num} & \end{array} \right\}$$

This generates the language of all well-formed formulas using addition and subtraction over binary numbers. For clarity, **Exp** denotes an arbitrary expression, **NumF** any number without leading zeroes and **Num** any number (possibly empty or with leading zeroes).

Those languages have less known formalisms, the Push-Down Automaton (again see [Rög23]) being the most common. For a characterisation of the context-free languages using logic, see section 3.4.2.

2.2.4 Context-Sensitive Languages

The most important category of languages for this work have multiple restrictions on the grammars which produce the same set.

One restriction is that all rules are of the form $\alpha\beta\gamma \rightarrow \alpha\varphi\gamma$ with $\alpha, \gamma \in (\Sigma \cup V)^*$, $\beta \in V$ and $\varphi \in (\Sigma \cup V)^+$. Additionally, if S is the start variable and never occurs on the right-hand side of any rule, we may include $S \rightarrow \varepsilon$.

Equivalently, we can have all grammars with $u \leq v$ for any rule $u \rightarrow v$, in addition to the special case with the start variable mentioned above. These grammars are called noncontracting.

The last, most useful form for proofs is the Kuroda normal form [Pet22], where all rules have one of the following forms:

- $A \rightarrow BC$
- $AB \rightarrow CB$

- $A \rightarrow a$
- $S \rightarrow \varepsilon$ if S is the start symbol and does not occur on any right-hand side

where $A, B, C, S \in V$ and $a \in \Sigma$.

Example 2.4. Consider the grammar $\langle \{S, B\}, \{a, b, c\}, R, S \rangle$ with

$$R = \left\{ \begin{array}{ll} S \rightarrow abc, & S \rightarrow aSBc, \\ cB \rightarrow Bc, & bB \rightarrow bb \end{array} \right\}$$

It generates the language $a^n b^n c^n$ for $n \in \mathbb{N}_1$ and is noncontracting.

The corresponding formalism for these languages are the linearly bounded nondeterministic Turing machines which can only write on the tape cells that contained a non-blank symbol. This and an equivalent extension of Second-Order logic will be proven in section 3.4.3.

2.2.5 Recursive Languages

The recursive languages are the most general languages in the hierarchy, as they don't have any restrictions on the rules. It can be shown that this set of languages is equivalent to the languages recognisable by a turing machine. By the Church-Turing thesis, this means that these are exactly the languages that can be computed by any of our computers and algorithms. Thus, we have a huge number of equivalent formalisms, including a RAM machine, while-programs and lambda calculus.

It is worth noting that there are languages which are not recursive. One of the most important example of these languages is the set of all (descriptions) of turing machines which halt on every input, also known as the halting problem. For the characterisation using logic, again refer to section 3.4.4.

3. Descriptive Complexity

3.1 Aims

In mathematics, abstraction is one of the most important tools as it enables us to make general statements and prove them for all the concrete instantiations of a concept. Formal Logic takes this even further and makes it possible to abstract mathematical thought itself. In Computer science, we are often interested in the amount of resources needed to compute a certain function or solve a certain problem, speaking in terms of time and storage space. Different forms of logic have the power to describe different types of problems. By focusing on decision problems¹, we can say a corresponding logical characterisation of a problem is a formula φ which is true if and only if a structure satisfies the required properties. By looking at the complexity of formulas which are needed to describe problems in terms of relations, operators, variables and other metrics, we can often find remarkably natural classes of logic corresponding to classes of problems.

Using these results, many insights into the underlying structure of real-world problems can be made which in turn can give us better ways to deal with them. Further, descriptive complexity has applications in database theory and computer aided verification and proofs.

3.2 Tools

As always, we first need to present some tools and techniques which will be used later in the proofs. Definition are again taken and modified from [Rög23] and [Imm99].

3.2.1 Complexity Theory

Complexity theory is the study of the resources, measured mostly in time and space, needed to compute certain problems². Also, we do not really care about constants in the computation, and thus use a notation which omits these.

Definition 3.1 (Big-O notation). Let f, g be functions $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$.

We say that $f \in \mathcal{O}(g)$ if there exists positive integers n_0, c such that for all $n \geq n_0$ we have

$$f(n) \leq c \cdot g(n)$$

¹Any problem can be reduced to boolean queries, for example by having a boolean query meaning "the i^{th} bit of an encoding of the answer is 1"

²The specific model of computation is not important as all give almost the same results. We will assume turing machines

Complexity classes can then be defined as all the problems which have a turing machine satisfying some bounds that can compute their solutions. Now we will define some common and important complexity classes.

Definition 3.2 (DTIME[$\mathcal{O}(t)$]). We say that a decision problem is in DTIME[$\mathcal{O}(t)$] if there exists a deterministic turing machine that takes a maximum of $f(n)$ steps on any input of size n and $f \in \mathcal{O}(t)$.

Definition 3.3 (P). We say that a decision problem is in P if there exists a polynomial q such that the problem is in DTIME[$\mathcal{O}(q)$]

Definition 3.4 (DSpace[$\mathcal{O}(t)$]). We say that a decision problem is in DSpace[$\mathcal{O}(t)$] if there exists a deterministic turing machine that visits a maximum of $f(n)$ tape cells on any input of size n and $f \in \mathcal{O}(t)$.

Definition 3.5 (PSPACE). We say that a decision problem is in PSPACE if there exists a polynomial q such that the problem is in DSpace[$\mathcal{O}(q)$]

We can go do the same for nondeterministic turing machines, and get the corresponding complexity classes NTIME, NP, NSPACE, NPSpace. There, we always take the maximum of tape cells and steps over any computation branch.

The complexity class P has a special meaning for computer scientists as these are the problems which are deemed "feasible" on modern computers.

3.2.2 Reduction and Completeness

A reduction can informally be seen as a method of using a problem we already solved to solve a new problem by converting this new problem into an instance of the old problem. These reduction can be very usefull to define complete problems for complexity classes, which in turn enable us to prove theorems for all problems of a specific complexity class.

Definition 3.6 (first-order reduction). Let \mathcal{C} be a complexity class and A and B be two problems over vocabularies σ and τ . Now suppose that there is some first-order query $I : \text{STRUC}[\sigma] \rightarrow \text{STRUC}[\tau]$ for which we have the following property:

$$\mathcal{A} \in A \Leftrightarrow I(\mathcal{A}) \in B$$

Then I is a first order reduction from A to B , denoted as $A \leq_{fo} B$.

First order reductions can then be used to show that some problem is also a member in some complexity class, as in most complexity classes, we can compute the first order query, and then we are left with a problem that we already know is in the required class. The converse can also be shown: for some problem B which is not in some complexity class \mathcal{C} , if we have $B \leq_{fo} A$, then A is also not in \mathcal{C} , as otherwise B would also be in \mathcal{C} , which is a contradiction.

Using the reductions, we can define completeness.

Definition 3.7 (Completeness via first-order reductions for Complexity Class \mathcal{C}). We say some problem A is complete for \mathcal{C} via \leq_{fo} if and only if

- $A \in \mathcal{C}$
- for all $B \in \mathcal{C}$, we have $B \leq_{fo} A$

Informally, a complete problem captures the essence of the complexity class. Further, they have an application in some proofs of equivalences between complexity classes \mathcal{C} and logics \mathcal{L} . These proofs follow the following steps as in [Imm99]:

1. Show that $\mathcal{L} \subseteq \mathcal{C}$ by providing a way to convert any formula $\varphi \in \mathcal{L}$ into an algorithm in \mathcal{C} .
2. Find a complete problem T for \mathcal{C} via first-order reductions.
3. Show that \mathcal{L} is closed under first-order reductions, that is that any formula can be extended by first-order quantifiers and boolean connectives and stay in \mathcal{L} .
4. Find a formula for T in \mathcal{L} , which shows $T \in \mathcal{L}$.

The above steps work, as for any problem B in \mathcal{C} , there is a first-order reduction I to T , and both \mathcal{L} and \mathcal{C} are complete via these reductions, so we also have $B \in \mathcal{L} = \mathcal{C}$.

3.2.3 Ehrenfeucht-Fraïssé Games

Ehrenfeucht-Fraïssé games are combinatorial games which are equivalent to first-order formulas and their extensions. Using these games, it is often possible to show non-expressability results for certain problems in some logic \mathcal{L} .

As a motivation, we can look at what it means for a formula to hold on some structure. Assume the formula has the form $\forall x \varphi(x)$. Then this can be seen as some opponent choosing some element $a \in |\mathcal{A}|$ and us now needing to show that $\varphi(a)$ holds. The case where the formula has the form $\exists x \psi(x)$ can be treated similarly, but we can choose the element ourselves.

Now for the formal definition

Definition 3.8 (Ehrenfeucht-Fraïssé Game). The k -pebble Ehrenfeucht-Fraïssé Game \mathcal{G}_k is played by two players: the Spoiler and the Duplicator on a pair of structures \mathcal{A} and \mathcal{B} using k pairs of pebbles. In each move, the spoiler places one of the remaining pebbles on an element of one of the two structures. Then, the duplicator tries to match the move on the other structure by placing the corresponding pebble on an element. We say that the duplicator wins the k -pebble Ehrenfeucht-Fraïssé Game on \mathcal{A}, \mathcal{B} if after the k rounds, the map $i : |\mathcal{A}| \rightarrow |\mathcal{B}|$ defined as for all elements of $|\mathcal{A}|$ with a pebble and the constants as the element in $|\mathcal{B}|$ with the corresponding pebble or constant forms a partial isomorphism. A partial isomorphism is an isomorphism formed for some subset of the universe, with all relations restricted to that subset.

In this context, the spoiler wants to show that \mathcal{A} and \mathcal{B} are different, whereas the duplicator wants to show their equivalence.

As this is a zero-sum game of full information, one of the two players must have a winning strategy. It can be proven that if the duplicator has a winning strategy for the k -pebble Ehrenfeucht-Fraïssé on \mathcal{A} and \mathcal{B} if and only if \mathcal{A} and \mathcal{B} agree on all formulas with less or equal to k nested quantifiers. We

can use these facts to prove non-expressibility of some problems in first-order logic by exhibiting two structures \mathcal{A}_k and \mathcal{B}_k for each k with one satisfying the problem constraints and the other not and a winning strategy for the duplicator on these two structures. This methodology can be extended to other logics by adding new moves or restrictions to the game.

3.3 Important Results

3.3.1 $\text{NSPACE}[s(n)] \subseteq \text{DSpace}[s(n)^2]$

3.3.2 SPACE Hierarchy theorem

The SPACE hierarchy theorem states that for both nondeterministic and deterministic space, we have problems that can be solved in some space $t(n)$, but not in less. Formally, we have

$$\text{DSpace}[o(t)] \subsetneq \text{DSpace}[\mathcal{O}(t)]$$

where $o(t)$ is the set of functions f such that $f \in \mathcal{O}(t)$ but $t \notin \mathcal{O}(f)$, that is all functions that grow more slowly than t . This holds for all (space constructible) $t \geq \log n$. The same holds for NSPACE.

We will present a proof for deterministic space.

Proof. The proof uses a diagonalization argument by presenting some machine D that takes a turing machine M and an input size in unary as input and does the opposite of M if it halts. We want to show that for all M which run in space $f(n) \in o(t(n))$, we have an input on which D and M do not agree. This would show that the language computed by D is not in $\text{DSpace}[o(t)]$, and thus the strict containment.

On input $\langle M, 1^k \rangle$ our machine D marks off $t(|\langle M, 1^k \rangle|)$ tape cells, which are the cells that are allowed for the computation. Further we also maintain a counter with size $|M| \cdot 2^{t(|\langle M, 1^k \rangle|)}$, which is the maximum amount of different configurations a TM can pass before looping on a binary tape of size $t(|\langle M, 1^k \rangle|)$. Then, we simulate M on input $\langle M, 1^k \rangle$. If we transcend any bound, we reject. For all M in $\text{DSpace}[o(t)]$, there is a k such that $f(n) \leq t(n)$ by definition. On this input, the simulation finishes, and we can invert the output.

This directly gives us an input for which M and D differ, and thus proves our claim. Furthermore, D runs in $\text{DSpace}[\mathcal{O}(t)]$ as by construction we assured that we do not run infinitely and that we stay within the space bound. \square

3.4 Results concerning the Chomsky hierarchy

3.4.1 Regular Languages

3.4.2 Context-Free Languages

3.4.3 Context-Sensitive Languages

3.4.4 Recursive Languages

3.5 Open questions

The domain of descriptive complexity is full of open questions as the proofs of lower bounds seems to be very difficult in most cases. Further, even separation between complexity classes which seem to take an exponential amount of resources compared to another one in practice can not be shown to be different.

3.5.1 $P \stackrel{?}{=} NP$

The P vs. NP question is the most emblematic question in descriptive complexity theory. In practice, for any NP-complete problem, only exponential worst-case algorithms are known. This leads to the widely believed conjuncture that $P \neq NP$. The problem is one of the seven Millennium Problems and a solution of equality or inequality is worth 1 Million US dollars. The consequences of a solution stating that $P = NP$ could have many practical advantages if it was constructive and had a low constant, as many important problems in research and logistics could be solved quickly. It would also mean the breakdown of most of modern cryptography, which relies on problem being intractable. On a conceptual level, it would mean that finding a proof to a problem is not harder than verifying its correctness, which would greatly impact the work of mathematicians. If a proof of the contrary would be known, this would focus the research more on the average case complexity of NP problems, but because of the continued lack of success on the question, this shift has already widely taken place.

3.5.2 $NSPACE[O(n)] \stackrel{?}{=} DSPACE[O(n)]$

4. Personal Contribution

5. Results

6. Conclusion and Direction

Thanks

List of Figures

Listings

Bibliography

- [HR22] Malte Helmert and Gabriele Röger. *Lecture: Discrete Mathematics in Computer Science*. University of Basel. 2022. URL: <https://dmi.unibas.ch/en/studies/computer-science/courses-in-fall-semester-2022/lecture-discrete-mathematics-in-computer-science/> (visited on 26/06/2024).
- [Imm99] Neil Immerman. *Descriptive Complexity*. Springer New York, 1999. ISBN: 9781461205395. DOI: 10.1007/978-1-4612-0539-5.
- [Pet22] Alberto Pettorossi. ‘Formal Grammars and Languages’. In: *Automata Theory and Formal Languages*. Springer International Publishing, 2022, pp. 1–24. ISBN: 9783031119651. DOI: 10.1007/978-3-031-11965-1_1.
- [Rög23] Gabriele Röger. *Lecture: Theory of Computer Science*. Universität Basel. 2023. URL: <https://dmi.unibas.ch/de/studium/computer-science-informatik/lehrangebot-fs23/main-lecture-theory-of-computer-science-1/> (visited on 26/06/2024).
- [Str94] Howard Straubing. *Finite automata, formal logic, and circuit complexity*. eng. Progress in theoretical computer science. Boston: Birkhäuser, 1994. ISBN: 9780817637194.

A. Mathematical Background

The definitions are taken from the lectures Discrete Mathematics in Computer Science [HR22] and Theory of Computer Science [Rög23] and also from the book Descriptive Complexity [Imm99].

A.1 Set Theory

Set An unordered collection of distinct elements, written with curly braces $\{\}$

Tuple An ordered collection of elements written with pointed braces $\langle \rangle$

Set operations There are multiple ways to form new sets from already existing sets:

Union denoted as \cup , an element is in $A \cup B$ if and only if it is in A or B

Intersection denoted as \cap , an element is in $A \cap B$ if and only if it is in A and B

Cartesian product denoted as \times , $A \times B$ is the set of tuples with an element of A and an element of B

Cartesian power A^k denotes the cartesian product of A with itself repeated k times

A.2 First Order Logic

We abbreviate first order logic as FO.

Variable A variable is an element that can have a value from a set.

Universe The set over which variables and constants can range

Relation A relation of arity k , $R(x_1, \dots, x_k)$ can be either true or false for any k -tuple of variables. In this work we always consider equality($=$), an ordering relation \leq , and $BIT(x, y)$, which means that the y^{th} bit of x is set in binary notation, to exist.

Vocabulary A tuple $\tau = \langle R_1^{a_1}, \dots, R_r^{a_r}, c_1, \dots, c_s \rangle$ of relations R_i with arity a_i and constants c_j (We omit functions as they can be simulated by a relation in our case)

Structure A tuple $\mathcal{A} = \langle |\mathcal{A}|, R_1^{\mathcal{A}}, \dots, R_r^{\mathcal{A}}, c_1^{\mathcal{A}}, \dots, c_s^{\mathcal{A}} \rangle$ where $|\mathcal{A}|$ is the universe, the constants are assigned a value from $|\mathcal{A}|$ and the truth of the relations have a truth value for each a_i -tuple from $|\mathcal{A}|^{a_i}$

First Order Formula A first order formula is inductively defined as follows:

Atoms Any formula of the form $R(x_1, \dots, x_k)$ for some relation of arity k is called an atomic formula

conjunction If φ and ψ are formulas, $(\varphi \wedge \psi)$ is a formula

disjunction If φ and ψ are formulas, $(\varphi \vee \psi)$ is a formula

negation If φ is a formula, $\neg\varphi$ is a formula

Existencial Quantification If φ is a formula, $\exists x\varphi$ is a formula

Universal Quantification If φ is a formula, $\forall x\varphi$ is a formula

Semantics For any structure, we can assign a truth value to any formula (by assigning values from the universe to free variables if they exist in the formula). We say \mathcal{A} satisfies ϕ (where ϕ is taken over the vocabulary of \mathcal{A}), denoted $\mathcal{A} \models \phi$ if and only if ϕ is true under the interpretation of the constant and relations of \mathcal{A} . This is inductively defined as follow:

Atoms For a formula ϕ of the form $R(x_1, \dots, x_k)$, we have $\mathcal{A} \models \phi$ if and only if the interpretation of the relation maps $\langle x_1, \dots, x_k \rangle$ to true

conjunction We have $\mathcal{A} \models (\varphi \wedge \psi)$ if and only if $\mathcal{A} \models \varphi$ and $\mathcal{A} \models \psi$

disjunction We have $\mathcal{A} \models (\varphi \vee \psi)$ if and only if $\mathcal{A} \models \varphi$ or $\mathcal{A} \models \psi$

negation We have $\mathcal{A} \models \neg\varphi$ if and only if $\mathcal{A} \not\models \varphi$

Existencial Quantification We have $\mathcal{A} \models \exists x\varphi$ if and only if there exists a $y \in |\mathcal{A}|$ such that $\mathcal{A} \models \varphi(y)$ (where $\varphi(y)$ denotes φ with any occurrence of x replaced with the element y)

Universal Quantification We have $\mathcal{A} \models \forall x\varphi$ if and only if for all $y \in |\mathcal{A}|$ we have $\mathcal{A} \models \varphi(y)$

First-Order Querries A first order query is a map from structures over one vocabulary σ to structures from another vocabulary τ . The mapping is done in such a way that we have first-order formulas for the universe (which is a subset of $|\mathcal{A}|^k$ for some k), the relation symbols and all the constants. For a more formally thorough definition see [Imm99]

Isomorphism An isomorphism is a map $i : |\mathcal{A}| \rightarrow |\mathcal{B}|$ with \mathcal{A}, \mathcal{B} over the same vocabulary which satisfies the following properties:

- i is bijective
- for every available relation R of arity a and every a -tuple \bar{e} in $|\mathcal{A}|^a$, we have

$$R^{\mathcal{A}}(e_1, \dots, e_a) \Leftrightarrow R^{\mathcal{B}}(i(e_1), \dots, i(e_a))$$

- for every constant symbol c , we have $i(c^{\mathcal{A}}) = c^{\mathcal{B}}$.

We write $\mathcal{A} \cong \mathcal{B}$

A.3 Second Order Logic

In second order logic, we extend the capabilities of first order logic with the ability to quantify over relations. We thus also need to extend our definitions. We abbreviate second order logic as SO.

SO variables A relation that is not given in the vocabulary and can be substituted with a specific interpretation

SO formula In addition to the inductive rules from the FO formulas, we can quantify over second order formulas

SO Existencial Quantification If φ is a formula, then $\exists V\varphi$ is a formula

SO Universal Quantification If φ is a formula, then $\forall V\varphi$ is a formula

SO Semantics Here we also need to extend the FO semantics

SO Existencial Quantification We have $\mathcal{A} \models \exists V\varphi$ if and only if there exists a relation U over $|\mathcal{A}|$ such that $\mathcal{A} \models \varphi(U)$ (where $\varphi(U)$ denotes φ with any occurrence of V replaced with U)

SO Universal Quantification We have $\mathcal{A} \models \forall V\varphi$ if and only if for all relations U over $|\mathcal{A}|$ we have $\mathcal{A} \models \varphi(U)$

A.4 Turing Machines

Turing machines are the most common model of computation. We abbreviate Turing Machines as TM

Informal definition A turing machine is a automaton with a finite number of states and an infinite tape. Using a read/write head, which can read one symbol on the tape, modify one symbol on the tape and move left and right, a Turing Machine can compute functions

Formal definition Formally, a Turing machine is a 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject} \rangle$, where

Q is the set of states

Σ is the set representing the symbols of which the input word on the tape can consist

Γ is the set of symbols which can be written or read on the tape

δ is the transition function, with $\delta : \Gamma \times Q \rightarrow \Gamma \times Q \times \{L, R\}$. So when a TM is in state n and reads a on the tape, δ tells us to which state we should transition, which symbol we should write and which direction we should move the read/write head

q_0 the start state

q_{accept} the accept state

q_{reject} the reject state

Turing computation At the beginning, the TM is in the start state, the input is written in a consecutive way on the tape and the read/write head is on the first character of the input word. In consecutive steps, the machine state then changes according to the transition function. If at some point the machine enters the accept or the reject state, the computation halts, and the TM is said to have accepted / rejected the input. In this work we will ignore the tape content after the computation and focus on decision problems.

Decidability If a TM halts on all inputs, we say that it decides a problem, as we can always be sure that the machine will accept or reject an input in finite time.

Nondeterministic TM (NTM) We can extend the transition function δ to allow multiple transitions from a given state. If there exists any computational path which leads to an accept state, the NTM accepts. This is not analog to how real sequential computers work, but allows interesting results, and is as powerfull as a normal deterministic TM.

Church-Turing Thesis According to the Church-Turing Thesis, this formalism is equivalent to what any computer can compute.

Independence declaration (German)

Ich, Yaël Arn, 4A

bestätige mit meiner Unterschrift, dass die eingereichte Arbeit selbstständig und ohne unerlaubte Hilfe Dritter verfasst wurde. Die Auseinandersetzung mit dem Thema erfolgte ausschliesslich durch meine persönliche Arbeit und Recherche. Es wurden keine unerlaubten Hilfsmittel benutzt. Ich bestätige, dass ich sämtliche verwendeten Quellen sowie Informanten/-innen im Quellenverzeichnis bzw. an anderer dafür vorgesehener Stelle vollständig aufgeführt habe. Alle Zitate und Paraphrasen (indirekte Zitate) wurden gekennzeichnet und belegt. Sofern ich Informationen von einem KI-System wie bspw. ChatGPT verwendet habe, habe ich diese in meiner Maturaarbeit gemäss den Vorgaben im Leitfaden zur Maturaarbeit korrekt als solche gekennzeichnet, einschliesslich der Art und Weise, wie und mit welchen Fragen die KI verwendet wurde. Ich bestätige, dass das ausgedruckte Exemplar der Maturaarbeit identisch mit der digitalen Version ist. Ich bin mir bewusst, dass die ganze Arbeit oder Teile davon mittels geeigneter Software zur Erkennung von Plagiaten oder KI-Textstellen einer Kontrolle unterzogen werden können.

Ort & Datum

Unterschrift
