Gymnasium Bäumlihof

Maturaarbeit

Theoretical Informatics: Formal languages and finite model thoery

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

29th July 2024, 4058 Basel

Contents

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 $a \to b$ 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 $y \to y'$ 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

$$R = \Big\{ S \to aSb, \quad S \to \varepsilon \Big\}$$

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

$$R = \begin{Bmatrix} S \to aO, & S \to \varepsilon, \\ O \to aS & \end{Bmatrix}$$

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 by a Deterministic finite automaton, which process the input word one character at a time

- 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 wich 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

$$R = \begin{Bmatrix} \mathbf{Exp} \to \mathbf{NumF}, & \mathbf{Exp} \to (\mathbf{Exp} + \mathbf{Exp}), \\ \mathbf{Exp} \to (\mathbf{Exp} - \mathbf{Exp}), & \mathbf{Exp} \to (-\mathbf{Exp}), \\ \mathbf{Num} \to 0\mathbf{Num}, & \mathbf{Num} \to 1\mathbf{Num}, \\ \mathbf{Num} \to \varepsilon, & \mathbf{NumF} \to 0, \\ \mathbf{NumF} \to 1\mathbf{Num} & \end{Bmatrix}$$

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 $\alpha\beta\gamma \to \alpha\varphi\gamma$ 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 $S \to \varepsilon$.

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

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

- $A \to BC$
- $AB \to CB$

- $A \to a$
- $S \to \varepsilon$ 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

$$R = \begin{Bmatrix} S \to abc, & S \to aSBc, \\ cB \to Bc, & bB \to bb \end{Bmatrix}$$

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 problems[1], 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 problems[2]. 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 $f, g : \mathbb{N} \to \mathbb{R}_+$.

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

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

[1]Any problem can be reduced to boolean queries, for example by having a boolean querry meaning "the ith bit of an encoding of the answer is 1"

[2]The 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 DTIME[$\mathcal{O}(t)$] 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 useful 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 $I : \text{STRUC}[\sigma] \to \text{STRUC}[\tau]$ 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 (Completness 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 inexpressibility 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 $i : |\mathcal{A}| \to |\mathcal{B}|$ 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 inexpressibility 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 NSPACE[$\mathcal{O}(s(n))$] $\subseteq$ DSPACE[$\mathcal{O}(s(n)^2)$]

This result is part of Savitch's Theorem, which introduces alternating turing machines in an intermediate step. These TMs are a generalisation of nondeterministic TMs, which can be seen as machine taking the "or" of all it's computation paths.

Definition 3.9 (Alternating Turing Machine)**.** An alternating Turing machine is a turing machine with two types of states: the existential and universal gates. Now, the acceptance conditions change compared to a NTM and depends on the state we are currently in. If we are in an existential state, we accept if and only if *at least one* of the computations leading on from this configuration is accepting. If we are in a universal state, we accept if and only *all* the computations leading on from this configuration are accepting.

These ATMs are now capable of also taking the "and" of the child states and maintain the ability to take the "or" of its children.

We define ATIME[$\mathcal{O}(t)$] and ASPACE[$\mathcal{O}(t)$] analogously to NTIME[$\mathcal{O}(t)$] and NSPACE[$\mathcal{O}(t)$].

Now we can proceed to the (quite technical) proof of Savitch's Theorem as presented in [Imm99].

Theorem 3.1 (Savitch's Theorem)**.** *For all space-constructible functions $t \geq \log n$ we have*

$$\mathit{NSPACE}[\mathcal{O}(t)] \subseteq \mathit{ATIME}[\mathcal{O}(t^2)] \subseteq \mathit{DSPACE}[\mathcal{O}(t^2)]$$

Proof. We start with the first inclusion, NSPACE[$\mathcal{O}(t)$] $\subseteq$ ATIME[$\mathcal{O}(t^2)$]. We now need to show that any NSPACE[$\mathcal{O}(t)$] turing machine can be simulated by a ATIME[$\mathcal{O}(t^2)$] alternating turing machine. Let N be a NSPACE[$\mathcal{O}(t)$] turing machine. Without loss of generality, we assume that N clears its tape after accepting and goes back to the first cell.

Now consider G_w, the computation graph of N on input w. We now see that N accepts w if and only if there is a path from the start configuration s to the accepting configuration t. We now present a routine $P(d, x, y)$ which asserts that there is a path of length at most 2^d from vertex x to y. Inductively, we can define P as follows:

$$P(d, x, y) = (\exists z)(P(d-1, x, z) \wedge P(d-1, z, y))$$

This formula asserts that there exists a middle vertex z for which there is a 2^{d-1} path from x to z and from z to y. Using an alternating turing machine, we can evaluate the formula using an existential state to find the middle vertex z, and then a universal state covering both shorter paths.

Now for the runtime analysis, we see that we need $\mathcal{O}(t(n))$ time to write down the middle vertex z, as a configuration includes the tape, which has length $\mathcal{O}(t(n))$. Further, we then need to evaluate some $P(d-1, z, y)$. By induction, we find that we need $\mathcal{O}(d \cdot t(n))$ time to compute $P(d, x, y)$. By the fact that there are only $2^{\mathcal{O}(t(n))}$ possible configurations, we get that the initial d is also in $\mathcal{O}(t(n))$, and thus our total runtime is $\mathcal{O}(t(n) \cdot t(n)) = \mathcal{O}(t(n)^2)$.

For the second inclusion, we need to simulate a ATIME[$\mathcal{O}(s(n))$] machine A using a DSPACE[$\mathcal{O}(s(n))$] machine (here, we substituted $s(n)$ for $t(n)^2$). Again, we consider the computation graph of A on input w. This graph has depth $\mathcal{O}(s(n))$ and size $2^{\mathcal{O}(s(n))}$.

We can systematically search this computation graph to get our answer. This is done by keeping a string of choices $c_1 c_2 \dots c_r$ of length $\mathcal{O}(s(n))$ made until this point. Note that this uniquely determines which state we are in.

Now, we can find the answer recursively. If we are in a halt state, we report this back to the previous state. In an existential state, we simulate its children, and if we get a positive result from one of them, we also return a positive result. In a universal state, we simulate its children, and if we get a positive result from all of them, we return a positive result.

In total, we use only $\mathcal{O}(s(n))$ space, to simulate A.

Thus, the second part of the theorem follows and by transitivity of $\subseteq$ we have NSPACE[$\mathcal{O}(t(n))$] $\subseteq$ DSPACE[$\mathcal{O}(t(n)^2)$]. □

We do not know if the containment is strict or not for any of the inclusions of the theorem. From this theorem, we also get the following interesting corollary.

Corollary 3.1.1. *We have PSPACE = NPSPACE.*

Proof.

$$\begin{aligned}
\text{NPSPACE} &= \bigcup_{k \in \mathbb{N}}^{\infty} \text{NTIME}[\mathcal{O}(n^k)] \\
&\subseteq \bigcup_{k \in \mathbb{N}}^{\infty} \text{DTIME}[\mathcal{O}(n^{2k})] \\
&= \bigcup_{k \in \mathbb{N}}^{\infty} \text{DTIME}[\mathcal{O}(n^k)] \\
&= \text{PSPACE} \\
&\subseteq \bigcup_{k \in \mathbb{N}}^{\infty} \text{NTIME}[\mathcal{O}(n^k)] \\
&= \text{NPSPACE}
\end{aligned}$$

□

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

$$\mathrm{DSPACE}[o(t)] \subsetneq \mathrm{DSPACE}[\mathcal{O}(t)]$$

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 DSPACE$[o(t)]$, and thus the strict containment.

On input $\langle M, 1^k \rangle$ our machine D marks of $t(|\langle M, 1^k \rangle|)$ 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 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 DSPACE$[\mathcal{O}(t)]$ as by construction we assured that we do not run infinitely and that we stay within the space bound. □

3.4 Results concerning the Chomsky hierarchy

Now that we have seen most of the required theory, we can start to apply it to the main theme of this work, the Chomsky hierarchy. For this section, we define the vocabulary on strings to be $\sigma = \langle \{0, \ldots, n-1\}, Q_a, Q_b, \ldots, Q_z, \leq, 0, 1, \max \rangle$. The universe consists of the numbers from 0 to $n-1$, we have a unary predicate for each character in Σ, a total ordering on the universe, and the constants $0, 1$ and $\max = n-1$.

3.4.1 Regular Languages

Here, we will show that the regular languages are captured exactly by second-order logic where we restrict ourselves to quantify only over predicates of arity one and do not include $\leq$. Further, we also are not allowed to use $\leq$, but have access to equality $x = y$ and the successor relation $x = y + 1$. We call this class SOM[+1].

First we need to present a formal definition of deterministic finite automata.

Definition 3.10 (DFA)**.** A deterministic finite automaton is a 5-tuple $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ where

Q is the set of states

Σ is the alphabet

δ is the transition function mapping a state and a symbol to the next state, so formally $\delta : Q \times \Sigma \to Q$.

q_0 the start state

F a subset of Q which are the accepting states.

We say that a DFA D accepts a word $w \in \Sigma^*$ if when starting at the start state, if we go through w and always transition to the next state according to the actual symbol in w and the actual state, we end up in an accepting state.

In [Rög23] and [Str94] there is a proof of the following fact we will use in our proof for SOM[+1]:

Theorem 3.2. *For any alphabet Σ, there is a DFA recognising language $L \subseteq \Sigma^*$ if and only if it is regular.*

Now we can start to prove our main theorem for regular languages.

Theorem 3.3. *For any alphabet Σ, a language $L \subseteq \Sigma^*$ is expressible in SOM[+1] if and only if it is regular.*

Proof. First we show that any regular language can be expressed in SOM[+1]. Let L be regular, end D_L be a DFA recognising the language. We assume L does not contain the empty word, otherwise we can recognise the language $L \setminus \{\varepsilon\}$ and then add $\varphi \vee \forall x(x \neq x)$, which adds the empty string back.

Now let D_L have k states. We can existentially quantify unary relations $X_1, \ldots, X_k$ to have the meaning that $X_i(y)$ is true if and only if D_L is in state i after y steps. Then, we need to make consistency checks. We present formulas for each of the consistency checks, and then can take the "and" of those to get our final formula $\exists X_1, \ldots, X_k(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$.

The start state is q_j We have

$$\varphi_1 \coloneqq \bigwedge_{i=1}^{k} (i = j \leftrightarrow X_j(0))$$

We end in an accepting state Let T_i be the set of all characters which lead from q_i to an accepting state. Then we have

$$\varphi_2 \coloneqq \bigwedge_{i=1}^{k} \left(X_i(\max) \to \bigvee_{a \in T_i} Q_a(\max) \right)$$

We move according to the transition function We have

$$\forall x \left(\forall y \left(y = x + 1 \to \left(\bigwedge_{i=1}^{k} \bigwedge_{a \in \Sigma} \Big((X_i(x) \wedge Q_a(x)) \to X_{\delta(i,a)}(y) \Big) \right.\right.\right.$$
$$\left.\left.\left. \wedge \bigwedge_{i=1}^{k} \bigwedge_{a \in \Sigma} \left((X_i(y) \wedge Q_a(x)) \to \bigvee_{r=1}^{k} (X_r(x) \wedge \delta(r, a) = j) \right) \right) \right) \right)$$

By induction, we can show that always exactly one i satisfies $X_i(x)$ for any x. Thus, if the created formula is satisfied, we know that D_L accepts the word, and thus we have described L in SOM[+1].

For the other direction, we need to introduce two new concepts.

One of them is the nondeterministic finite automaton, which is analogous to the nondeterministic turing machine as it can also have multiple transitions going from the same state. As with the NTM and the TM, both the DFA and the NFA have the same expressive power.

The other concept is that of $(\mathcal{V}_1, \mathcal{V}_2)$-structures. These structures are generalisations of our former vocabulary σ as they have characters in $A \times \mathcal{P}(\mathcal{V}_1) \times \mathcal{P}(\mathcal{V}_2)$. These structures are useful as we can make $\mathcal{V}_1$ to be the set of free first-order variables in a formula φ and $\mathcal{V}_2$ be the set of free second-order variables in the formula. If at a position i in our $(\mathcal{V}_1, \mathcal{V}_2)$-structures we have x in the first-order component of its character, we see this as meaning that $x = i$. For the second-order variables in the third component, an X at position i means that $X(i)$ holds.

Now, we can prove by induction that all formulas in SOM[+1] with free variables in $\mathcal{V}_1$ and $\mathcal{V}_2$ are regular. Sentences, the formulas without free variables are the special case where $\mathcal{V}_1, \mathcal{V}_2 = \emptyset$.

First, we need to check that the $(\mathcal{V}_1, \mathcal{V}_2)$-structures are consistent, and no first-order variable x appears more than once. This can be done by a NFA which has one state for each subset of variables, and extends its subset while going over the string. If a variable appears twice, we enter a state that always loops and rejects.

Then, we see that the atomic formulas can be checked, as $x = y$, $x = y + 1$ and $Q_a(x)$ are easy to check, and checking $X(x)$ is equivalent to looking if the occurrence of x has X in the third component. We always need to take the intersection with the NFA which checks if the structure is valid.

All boolean connective are also valid as regular languages are closed under complement, intersection and union as seen in [Rög23].

The most difficult case is a formula of the form $\exists x\varphi$ (as $\forall x\varphi \equiv \neg\exists x\neg\varphi$)). If $\exists x\varphi$ is over $(\mathcal{V}_1, \mathcal{V}_2)$-structures, then φ is over $(\mathcal{V}_1 \cup \{x\}, \mathcal{V}_2)$-structures. By induction, we know that φ defines a regular language and thus there is a NFA N which recognises it. For the new automaton, we duplicate our states, with the meanings "used x" and "not used x". If we are in a state where x was used, we can not take any transition with x in the second set. If we are in a state where x was not used, we can take a transition with x in the second set and go to the corresponding state with x used or take a transition where x is not used and go to the corresponding state where x was not used.

The remaining case with second-order variables is treated analogously, without the restriction on the number of times the variable is used, so we do not need to duplicate our states.

By induction, we have thus showed the other direction, and we see that SOM[+1] and the regular languages are equivalent. $\square$

3.4.2 Context-Free Languages

For the context-free languages, we will show that they have an underlying structure that includes matchings.

A matching relation is a binary relation M on the universe $\{0, \dots, n-1\}$ which has the following properties:

increasing If $M(i,j)$, then $i < j$

uniqueness Any k in the universe appears at most once in the relation

non-crossing If we were to draw arcs for the matching, none of them would intersect. Formally, if $M(i,j)$ and $M(k,l)$, then either $j < k$ or $l < i$

Visually, we can think of these relations as nested ranges over the universe.

With matchings, we can define FO(∃Match) as the first order logic extended with existential quantification over matching relations.

Theorem 3.4. *For any alphabet Σ, a language $L \subseteq \Sigma^*$ is expressible in FO(∃Match) if and only if it is context-free.*

For this, we first introduce a normal form for context-free grammars. The proof that every context-free grammar can be converted to this form can be found in [LST95].

Lemma 3.5. *Every context free language has a grammar which satisfies*

- *All rules are of one of the two forms*
  - $S \to \alpha$ *with* $\alpha \in \Sigma$
  - $X \to \alpha u \beta$ *with* $\alpha, \beta \in \Sigma$ *and* $u \in (\Sigma \cup V)^*$
- *For all production rules with a right hand side that has at least one nonterminal we define its* pattern. *The pattern of rule* $X \to v_0 X_1 v_2 X_2 \dots X_s v_s$ *is defined to be* $v_0|v_1|\dots|v_s$, *where* $|$ *is a new symbol not in* Σ *We require that for any two rules with the same pattern, they have the same left-hand side and thus the source nonterminal can be uniquely identified by the pattern.*

For a specific arch $\langle i,j \rangle$ in a matching M on some word structure, we can also determine a pattern. For this, we go through all indices from i to j and add the character at the actual position. If we are at a starting point of some arch $\langle k,l \rangle$, instead of adding the actual character, we add $|$ and continue at $l+1$.

Now, we can start with the proof of our theorem.

Proof. We want to find a formula such that for all archs $\langle i,j \rangle$ in M, the substring from i to j can be derived from the non-terminal for which we have a rule with the pattern of $\langle i,j \rangle$.

For this, we say that arch $\langle i,j \rangle$ *corresponds* to a production rule p if they have the same pattern. For any string u, we have a formula $\phi_u(i,j)$ which means that the substring from i to j is u. This formula is easy to write as we only need to check each character one by one using the successor relation. We can express correspondence to $p \equiv X \to v_0 X_1 v_2 X_2 \dots X_s v_s$ (including rules with terminal right-hand side) by the following formula:

$$\begin{aligned}\mathcal{X}_p(x,y) \equiv \exists x_1, y_1, \dots, x_s, y_s\, &((x < x_1 \wedge x_1 < y_1 \wedge y_1 < x_2 \wedge \dots \wedge y_s < y) \wedge \\ &(\phi_{v_0}(x, x_1 - 1) \wedge \phi_{v_1}(y_1 + 1, x_2 - 1) \wedge \dots \wedge \phi_{v_s}(y_s + 1, y)) \wedge \\ &(M(x_1, y_1) \wedge \dots \wedge M(x_s, y_s) \wedge M(x,y)) \wedge \\ &\forall k, l\, (M(k,l) \to ((x \le k \wedge l \le y) \vee (x_1 \le k \wedge l \le y_1) \vee \dots \vee (x_s \le k \wedge l \le y_s))))\end{aligned}$$

The first line means that the v_i are in the right order, the second that they do correspond, the third that there are arches between the v_i and the last that there are no other arches.

Now, we want to be more general, and express the that the pattern of $\langle i, j\rangle$ corresponds to some production rule with left-hand side X. Let $\widetilde{\mathcal{X}}_X(x, y)$ be the disjunction of all $\mathcal{X}_p$ with left-hand side X. Because our normal form says that the pattern uniquely determines the left-hand side, for each arch which has arches underneath, we have a unique corresponding nonterminal.

Now, we want to have a formula which expresses not only correspondence for a production rule p, but also that the nonterminals are correct. For this, we supplement our formula $\mathcal{X}_p$ with a new line, expressing that the nonterminals correspond.

$$\begin{aligned}\widetilde{\mathcal{X}}_p(x,y) \equiv \exists x_1, y_1, \dots, x_s, y_s\, (&(x < x_1 \wedge x_1 < y_1 \wedge y_1 < x_2 \wedge \cdots \wedge y_s < y) \wedge \\ &(\phi_{v_0}(x, x_1 - 1) \wedge \phi_{v_1}(y_1 + 1, x_2 - 1) \wedge \cdots \wedge \phi_{v_s}(y_s + 1, y)) \wedge \\ &(M(x_1, y_1) \wedge \cdots \wedge M(x_s, y_s) \wedge M(x, y)) \wedge \\ &\forall k, l\, (M(k, l) \to ((x \le k \wedge l \le y) \vee (x_1 \le k \wedge l \le y_1) \vee \cdots \vee (x_s \le k \wedge l \le y_s))) \\ &\left(\widetilde{\mathcal{X}}_{X_1}(x_1, y_1) \wedge \cdots \wedge \widetilde{\mathcal{X}}_{X_s}(x_s, y_s)\right)\Big)\end{aligned}$$

Finally, with P being our set of rules, we have a formula that tells us a word can be derived from the start symbol:

$$\exists M \left(\widetilde{\mathcal{X}}_S(0, \max) \wedge \forall x, y \left(M(x, y) \to \bigvee_{p \in P} \widetilde{\mathcal{X}}_p(x, y) \right) \right)$$

Now, by construction, if and only if a word satisfies the formula, there is a derivation from S for the word, as each arch can be seen as a derivation step, which we check is valid with our formula. We used our assumption from the normal form to show that the $\widetilde{\mathcal{X}}_p$ are only satisfied when the arches which are nonterminal do not belong to any other nonterminal symbol then the one in the rule. If two terminal rules coincide, we do not care as no further derivation is possible.

Now, we come to the other direction. This direction requires some new notation and lemmas. We will use the notion of tree languages.

Definition 3.11 (Tree Language)**.** In a tree language, we have a rooted tree, which has an order on its vertices in leftmost depth-first way[3]. Each node has a label, and each label has an arity which corresponds to the out-degree of the node. A tree language is the set of all trees over a finite label set.

We define the *leaf alphabet* of a tree language to be the set of 0-ary labels. For any tree T in a tree language, we define the *yield* of T to be the leaf labels concatenated according to the order relation from left to right.

The vocabulary of a tree language $\mathcal{T}$ is $\tau = \langle \{0, \dots, n-1\}, Q_a, Q_b, \dots, Q_z, \le, 0, 1, \max, C^2 \rangle$ In addition to the relations for each label, there is a child relation $C(i, j)$ which means "node i is a child of node j".

Now, we present two lemmas for recognisable tree languages and their relation to context-free

[3]The order which we find by doing a depth-first search on the tree, entering the leftmost node first

languages.

Lemma 3.6 ([MW67]). *A language $L \subseteq \Sigma^*$ is context free if and only if there is a recognisable tree language T with leaf alphabet Σ for which a word is in L if and only if it is the yield of some tree in T.*

Lemma 3.7 ([TW68]). *A tree language T is recognisable if and only if there is a monadic second order sentence that recognises it.*

We now present some relations that can be written in MSO on trees.

Lf(i) Node i is a leaf $\mathrm{Lf}(i) \equiv \forall x \neq C(x, i)$

Lc(i, j) Node i is the leftmost child of j $\mathrm{Lc}(i, j) \equiv C(i, j) \wedge \forall x(C(x, j) \rightarrow i < x)$

Rc(i, j) Node i is the rightmost child of j $\mathrm{Rc}(i, j) \equiv C(i, j) \wedge \forall x(C(x, j) \rightarrow x < i)$

An(i, j) Node i is an ancestor or j

$$\begin{aligned}\mathrm{An}(i, j) \equiv \exists U(U(i) \wedge U(j) \wedge \forall x(U(x) \rightarrow \\ ((x \neq i \leftrightarrow \exists y(C(x, y) \wedge U(y))) \wedge (x \neq j \leftrightarrow \exists y(C(y, x) \wedge U(y))))))\end{aligned}$$

Pt(U, i, j) Node i is an ancestor or j, j is a leaf and U contains all nodes in the path from i to j.

$$\begin{aligned}\mathrm{Pt}(U, i, j) \equiv U(i) \wedge U(j) \wedge \mathrm{Lf}(j) \wedge \forall x(U(x) \rightarrow \\ ((x \neq i \leftrightarrow \exists y(C(x, y) \wedge U(y))) \wedge (x \neq j \leftrightarrow \exists y(C(y, x) \wedge U(y)))))\end{aligned}$$

Using these two lemmas, we can continue by presenting for any formula φ in FO($\exists$Match) a formula ϕ in MSO over trees such that

$$w \models \varphi \Leftrightarrow \text{there exists a tree } T \text{ with yield } w \text{ such that } T \models \phi$$

For this, we present a class of trees which correspond to a word with a matching. For any word w with matching M, we can construct a tree over $\Sigma \cup \{\oplus^2, \odot^2\}$. We do this using an intermediate step. First, we construct a tree with wrong arity for nodes of type $\oplus$ by assigning one node of type $\oplus$ to each arch, with edges to every direct arch underneath it and every character directly underneath it. If we have multiple trees, we add a new $\odot$ node on top with an edge to all roots of these trees. To fix the arity issue, we repeat the following procedure until there are no nodes with more than outdegree 2.

Take some node with outdegree greater than 2. Take the two leftmost children and add a $\odot$ node with an edge to both, and an edge from the new node to the former parent. This procedure will eventually terminate as we always decrease the outdegree of some node by one and add a valid node.

We can see that this procedure can also be done backward if and only if for any node of type $\oplus$, the leaf on the leftmost and rightmost path of a node are distinct and no $\oplus$ node occurs on the path between the two. We can express this property of a tree by the following formula.

$$\begin{aligned}\Upsilon \equiv \forall x(Q_{\oplus}(x) \to (\exists y, z, U_y, U_z(y \neq z \wedge \mathrm{Pt}(U_z, x, z) \wedge \mathrm{Pt}(U_y, x, y) \wedge \\ \forall r(U_y(r) \to ((r = y \vee \exists w(U_y(w) \wedge \mathrm{Rc}(w, r))) \wedge (r = y \vee r = x \vee Q_{\odot}(r)))) \wedge \\ \forall r(U_z(r) \to ((r = z \vee \exists w(U_z(w) \wedge \mathrm{Lc}(w, r))) \wedge (r = y \vee r = x \vee Q_{\odot}(r))))))\end{aligned}$$

Here, in the first line we assert that for every node with label $\oplus$, we have two distinct leaves y, z with paths U_y and U_z. The second and third line assert the same for x and y, that they are the rightmost / leftmost leaf and that no other $\oplus$ type node lies on the path from them to x.

Now, we want to convert our formula φ over strings with a matching to a formula γ over trees. Then we can assert that the tree represents a string with matching and the yield satisfies φ using $\Upsilon \wedge \gamma$.

For this, we need to restrict any quantifiers in φ to the leaves by replacing $\exists x \phi$ with $\exists x \mathrm{Lf}(x) \wedge \phi$ and $\forall x \phi$ with $\forall x \mathrm{Lf}(x) \to \phi$. Further, we replace $M(z, y)$ by

$$\begin{aligned}m(z, y) \equiv \exists x(Q_{\oplus}(x) \wedge (\exists U_y, U_z(y \neq z \wedge \mathrm{Pt}(U_z, x, z) \wedge \mathrm{Pt}(U_y, x, y) \wedge \\ \forall r(U_y(r) \to ((r = y \vee \exists w(U_y(w) \wedge \mathrm{Rc}(w, r))) \wedge (r = y \vee r = x \vee Q_{\odot}(r)))) \wedge \\ \forall r(U_z(r) \to ((r = z \vee \exists w(U_z(w) \wedge \mathrm{Lc}(w, r))) \wedge (r = y \vee r = x \vee Q_{\odot}(r)))))))\end{aligned}$$

which is very similar to Υ.

This is already everything we need to do, and thus we can conclude that

$$w \models \varphi \Leftrightarrow \text{there exists a tree } T \text{ with yield } w \text{ such that } T \models \Upsilon \wedge \gamma$$

Thus, we have proved both directions and see that the context free languages are exactly captured by FO($\exists$Match) ☐

3.4.3 Context-Sensitive Languages

This is the language class that interests us most, as it has be studied less extensively then other language classes. Nevertheless, there are some known formalisms. One of them is the linear bounded nondeterministic turing machine. The two directions of the proof were presented separately in [Kur64] and [Lan63].

Theorem 3.8. *The class of context-sensitive languages is exactly the class of languages accepted by a linear bounded nondeterministic turing machine.*

Proof. For the direction from grammar to turing machine, we only need to show that our turing machine can simulate a derivation backwards. We know that every context-sensitive language has a noncontracting grammar G. Using this fact, we can construct a nondeterministic turing machine N which scans the current tape and whenever it recognises a pattern of the right-hand side of a production rule in G, it decides whether it replaces it or not by the left-hand side of the rule. If some computation branch of N ends up with only the start symbol of G on the tape, we accept. Essentially,

N simulates a derivation of w from the start symbol backwards. Because we try all possibilities by nondeterminism, we know that if N does not accept w, there is no derivation ending in w from the start symbol of G. As we assumed G is noncontracting, replacing the right-hand side of a rule with the left-hand side never makes the word longer, and thus we only need $\mathcal{O}(|w|)$ space.

The other direction works by explicitly defining a grammar which simulates any linear bounded automaton backwards. Without loss of generality, we include an end marker $\#$.

Let $N = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\rangle$ be a NSPACE[$\mathcal{O}(n)$] turing machine. Then, we construct a grammar $G = \langle V, \Sigma, P, S\rangle$ with $V = \bigcup_{q_i \in Q} \bigcup_{a_j \in \Gamma} \{b_{q_i,a_j}\} \cup \{S, L, R, \#\} \cup \bigcup_{a_w \in \Gamma \setminus \Sigma} \{a_w\}$. The $b_{q,a}$ represent a position on the tape including the actual state and the actual character, in addition we also have the start state, the end marker and some utility nonterminals.

We now add the following rules to P:

- For each $a_i \in \Gamma$, we add a rule $S \to Lb_{q_{accept},a_i}R$ to P. These rules mean that we are in a final accept state. To extend the final tape, we add again for each $a_i \in \Gamma$ rules $L \to La_i$, $L \to \#$, $R \to a_iR$ and $R \to \#$ to our rule set. These rules allow derivations from S to an end tape of the form $\#a_{i_1} \dots b_{q_{accept},a_{i_j}} \dots a_{i_k}\#$.
- For each rule $\langle a_k, q_l, L\rangle \in \delta(q_i, a_j)$, we add rule $b_{a_w,q_l}a_k \to a_w b_{a_j,q_i}$ for every $a_w \in \Gamma$. Similarly, for each rule $\langle a_k, q_l, R\rangle \in \delta(q_i, a_j)$, we add rule $a_k b_{a_w,q_l} \to b_{a_j,q_i}a_w$ for every $a_w \in \Gamma$. We can clearly see that these rules simulate the NTM backwards.
- For the start, we include $\#b_{a_i,q_0} \to \#a_i$ for all $a_i \in \Sigma$ This rule allows us to say that we are at the start of our computation and "remove" the read-write head to get our initial input word.

As for any word in G we can follow back the derivation to an accept state, we have that both N and G define the same language.

Thus, we are done and have proven the equivalence of context-sensitive languages and linear bounded automata. $\square$

Now that we showed this equivalence, we will show that the context-sensitive languages are equivalent to the languages definable in MSO(TC), where we supplement monadic second order logic with a transitive closure operator.

For this, we first define the transitive closure of a formula.

Definition 3.12 (Transitive closure)**.** Let $\phi\left(\overset{k}{a}, \overset{k}{b}\right)$ be a formula with $2k$ free variables. We can see this formula as an edge relation over the graph with vertices $\overset{k}{c}$. Then, the transitive closure of the formula $\left(TC_{\overset{k}{a},\overset{k}{b}}\phi\left(\overset{k}{a}, \overset{k}{b}\right)\right)\left(\overset{k}{u}, \overset{k}{v}\right)$ is true if and only if there is a path in the $\overset{k}{c}$ graph from $\overset{k}{u}$ to $\overset{k}{v}$ using edges from ϕ. Equivalently, we can also define it to be the minimal relation such that if $R(x, y)$ and $R(y, z)$, the $R(x, z)$.

Theorem 3.9. *A language L is context-sensitive if and only if it can be described by a formula in $MSO(TC)$.*

Proof. We use theorem 3.8 and show the equivalence of MSO(TC) and NSPACE[$\mathcal{O}(n)$] Turing machines. Then the theorem will follow immediately.

First, we show that any formula in MSO(TC) can be evaluated by NSPACE[$\mathcal{O}(n)$] turing machine. The first step is to notice that any relation of MSO can be represented on the tape in $\mathcal{O}(n)$ space. So, for any sub-formula of the form $\left(TC_{\overset{k}{a},\overset{k}{b}}\phi\left(\overset{k}{a},\overset{k}{b}\right)\right)\left(\overset{k}{u},\overset{k}{v}\right)$, we write down $\overset{k}{u}$ and guess the next vertice. Then, we can check if the transition is valid by evaluating phi. We can repeat this process until we reach $\overset{k}{v}$. Because Immerman showed that NSPACE is closed under complementation in [Imm88], we know that there is also a NSPACE[$\mathcal{O}(n)$] turing machine that computes any formula of the form $\neg\left(TC_{\overset{k}{a},\overset{k}{b}}\phi\left(\overset{k}{a},\overset{k}{b}\right)\right)\left(\overset{k}{u},\overset{k}{v}\right)$. The other parts of the formula can be evaluated easily in linear space as we just need to write down relations when quantifying and otherwise remember in which part of the constant-size formula we are actually.

For the other direction, consider a NSPACE[$\mathcal{O}(n)$] Machine N. As in the previous proof with the grammars, we enrich our logic with new symbols b_{a_i,q_i} for all pairs of states and symbols, which means that we add a relation Q_b for all such b to our vocabulary.

Now consider a tuple $\overline{X} = \langle Q_{b_1}, \dots, Q_{b_r}\rangle$. This tuple completely represents an instantaneous configuration of the tape and the machine. We can now write a formula $\varphi(\overline{X},\overline{Y})$ which means that a transition from state $\overline{X}$ to $\overline{Y}$ is possible in N. This can be done by a big disjunction over all rules of N. As a transition a new character b_i is only determined by the actual character and the two characters left and right of it[4], we can write all transitions of N in the form $\langle b_k, b_l, b_m\rangle \to \langle b_i, b_j, b_w\rangle$. If P is the set of all transitions, we have

$$\varphi(\overline{X},\overline{Y}) \equiv \exists i\left(\forall j\left(|i-j|>1 \to \bigwedge_{b_i}\left(Y_{Q_{b_i}}(j) \leftrightarrow X_{Q_{b_i}}(j)\right)\right)\wedge\right.$$
$$\left.\bigvee_{\langle b_k,b_l,b_m\rangle\to\langle b_u,b_v,b_w\rangle\in P}\left(X_{Q_{b_k}}(i-1)\wedge X_{Q_{b_l}}(i)\wedge X_{Q_{b_m}}(i+1)\wedge Y_{Q_{b_u}}(i-1)\wedge Y_{Q_{b_v}}(i)\wedge Y_{Q_{b_w}}(i+1)\right)\right)$$

The first line asserts that apart from the indices where the head is and thus where we change something, the tape stays unchanged. The second line tells us that there exists a rule in P such that the characters in $\overline{Y}$ at positions $i-1, i, i+1$ follow from the previous characters in $\overline{X}$.

Now, we can take the transitive closure over φ, starting with the input relations, and ending at an accept position. Without loss of generality we can assume that N clears its tape after accepting, so the end position is unique. Now, our formula of the transitive closure over φ holds if and only if there is an accepting path in N starting at the input word w.

This concludes the equivalence of MSO(TC) and context-sensitive languages. □

An interesting normal form for MSO(TC) can be derived from the proof.

[4]we include end markers for this to work without modification for the ends of the tape

Corollary 3.9.1. *Every formula in MSO(TC) can be written in the form*

$$\left(TC_{\overline{X},\overline{Y}}\varphi(\overline{X},\overline{Y})\right)\left(\overline{U},\overline{V}\right)$$

Proof. We have given an explicit way to convert any formula into a turing machine and back in theorem 3.9. By construction this always gives us a formula of the required form. □

3.4.4 Recursive Languages

The most general case is interesting as it gives us a logical formalism for all problems which are computable at all.

The proof relies on Diophantine sets. Those sets are the sets that correspond to the tuples which have a solution for some Diophantine equation. A Diophantine equation is a polynomial equation P with a tuple $\overline{x}$ of parameters and a tuple $\overline{y}$ of variables. A tuple $\overline{x}$ has a solution if there exists a tuple $\overline{y}$ such that the $P(\overline{x},\overline{y}) = 0$. The famous MRDP Theorem states that the Diophantine sets are exactly the computable sets. At the same time, this shows that Hilberts 10^{th} problem is unsolvable. A full proof of these facts can be found in [Mat96].

With this new characterisation, there is a quite straightforward characterisation of computable sets. The logic FO($\exists\mathbb{N}$) consists of all formulas $\phi(\overline{x}) \equiv \exists\overline{y}(\varphi(\overline{x},\overline{y}))$ with φ having only bounded quantifiers (of the form $\exists x < y$ or $\forall x < y$), addition, multiplication, equality, and any constant natural number [Ent20]. The existential quantifiers at the beginning are allowed to range over all the natural numbers.

These formulas can define exactly the diophantine sets, as the formulas we present are exactly those that mean "there is a tuple $\overline{y}$ of natural numbers such that some polynomial is satisfied". Thus, they also define exactly all computable sets.

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 wich 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[$\mathcal{O}(n)$]$\stackrel{?}{=}$DSPACE[$\mathcal{O}(n)$]

This problem is known under the name first Linear bounded automaton problem since its proposal by Kuroda in [Kur64], and asks if nondeterminism adds power in the context of bounded space. This comes from the fact that a NSPACE[$\mathcal{O}(n)$] turing machine can be seen as a TM with a linear bound on its space usage. This theorem is of interest as we know that NSPACE[$\mathcal{O}(n)$] is equivalent to the context-sensitive languages by section 3.4.3.

Since the proposal, there were two advances. One is the proof that NSPACE is closed under complement. The contrary would have implied NSPACE[$\mathcal{O}(n)$] $\neq$ DSPACE[$\mathcal{O}(n)$] as DSPACE is closed under complement. The second advance is Savitch's Theorem in section 3.3.1 which already gives a bound for simulating NSPACE using DSPACE machines. It is not known if this theorem is optimal, that is whether the blowup by a power of 2 is optimal or if we can do better.

An equality would imply that the context-sensitive languages can be recognised by a deterministic linear bounded automaton, which could make recognising words in context-sensitive langauges easier and faster.

4. Personal Contribution

5. Results

6. Conclusion and Direction

Thanks

List of Figures

Listings

Bibliography

[Ent20] Rising Entropy. *The Arithmetic Hierarchy and Computability.* Aug. 2020. URL: https://risingentropy.com/the-arithmetic-hierarchy-and-computability/ (visited on 24/07/2024).

[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).

[Imm88] Neil Immerman. 'Nondeterministic space is closed under complementation'. In: *SIAM J. Comput.* 17.5 (Oct. 1988), pp. 935–938. ISSN: 0097-5397. DOI: 10.1137/0217058.

[Imm99] Neil Immerman. *Descriptive Complexity.* Springer New York, 1999. ISBN: 9781461205395. DOI: 10.1007/978-1-4612-0539-5.

[Kur64] S. Y. Kuroda. 'Classes of languages and linear-bounded automata'. In: *Information and Control* 7.2 (June 1964), pp. 207–223. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(64)90120-2.

[Lan63] Peter S. Landweber. 'Three theorems on phrase structure grammars of type 1'. In: *Information and Control* 6.2 (June 1963), pp. 131–136. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(63)90169-4.

[LST95] Clemens Lautemann, Thomas Schwentick and Denis Thérien. 'Logics for context-free languages'. en. In: *Computer Science Logic.* Ed. by Leszek Pacholski and Jerzy Tiuryn. Berlin, Heidelberg: Springer, 1995, pp. 205–216. ISBN: 9783540494041. DOI: 10.1007/BFb0022257.

[Mat96] Jurij V. Matijasevič. *Hilbert's tenth problem.* 3. print. Foundations of computing. Aus dem Russ. übers. Cambridge, Mass. [u.a.]: MIT Press, 1996. 264 pp. ISBN: 0262132958.

[MW67] J. Mezei and J. B. Wright. 'Algebraic automata and context-free sets'. In: *Information and Control* 11.1 (July 1967), pp. 3–29. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(67)90353-1.

[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.

[TW68] J. W. Thatcher and J. B. Wright. 'Generalized finite automata theory with an application to a decision problem of second-order logic'. en. In: *Mathematical systems theory* 2.1 (Mar. 1968), pp. 57–81. ISSN: 1433-0490. DOI: `10.1007/BF01691346`.

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
>
> **Power set** denoted as $\mathcal{P}(A)$ contains all subsets of A

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, $R(x_1, \ldots, x_k)$ 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 $\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$ 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 $\mathcal{A} = \langle |\mathcal{A}|, R_1^{\mathcal{A}}, \ldots, R_r^{\mathcal{A}}, c_1^{\mathcal{A}}, \ldots, c_s^{\mathcal{A}} \rangle$ 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 $R(x_1, \ldots, x_k)$ for some relation of arity k is called an atomic formula

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

disjuction 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 $R(x_1, \ldots, x_k)$, we have $\mathcal{A} \models \phi$ if and only if the interpretation of the relation maps $\langle x_1, \ldots, x_k \rangle$ to true

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

disjuction 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 $\mathcal{A} \models \varphi(y)$

Free variables A variable is called free if there is an occurrence of it which is not bound by a quantifier whose scope surrounds it

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 $i : |\mathcal{A}| \to |\mathcal{B}|$ 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 $\overline{e}$ in $|\mathcal{A}|^a$, we have

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

- 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 abreviate 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 abreviate 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 $\delta : \Gamma \times Q \to \Gamma \times Q \times \{L, R\}$. 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.

Nondetermenistic TM (NTM) We can extend the transition function δ to allow multiple transitions from a given state. Formally, we then have $\delta : \Gamma \times Q \to \mathcal{P}(\Gamma \times Q \times \{L, R\})$. 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.

Space/Time-Constructible functions A function $f(n)$ is time constructible if there exists a TM which on input 1^n writes $f(n)$ in binary on its tape in time $f(n)$. Space-constructible are analogous.

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