Regular languages defined by first-order formulas without quantifier alternation

Andreas Krebs, University of Tübingen
Howard Straubing, Boston College

August 23, 2022

Abstract

We give a simple new proof that regular languages defined by first-order sentences with no quantifier alternation can be defined by such sentences in which only regular atomic formulas appear. Earlier proofs of this fact relied on arguments from circuit complexity or algebra. Our proof is much more elementary, and uses only the most basic facts about finite automata.

1 Introduction

1.1 Background

This paper concerns the use of formulas of first-order logic to define properties of strings over a finite alphabet $A$. The description of how these formulas work will be rather informal in this introductory section, but we will be more precise later on. Variables in our formulas are interpreted as positions in a string: these should be thought of as positive integers, where the leftmost position in a string is 1. The formulas themselves are built from a base consisting of two types of atomic formulas: The first type consists of unary predicates $a(x)$, where $a \in A$, which we interpret to mean that the letter in position $x$ is $a$. The other atomic formulas we call ‘numerical predicates’. What makes them numerical is that they depend only on the numerical values of positions and the length of the string, and not on the letters in those positions. So, for example, $x < y$ is a binary numerical predicate, asserting that the position represented by the variable $x$ is strictly to the left of that represented by the position $y$. ‘$x$ is prime’ is a unary numerical predicate. ‘The length of the string is the sum of two cubes’ is a 0-ary numerical predicate. An example of a sentence (i.e., a formula without free variables) in this logic is

$$\forall x \forall y ((a(x) \land b(y)) \rightarrow x|y).$$

\footnote{We use ‘string’ and ‘word’ interchangeably.}
This says that every position containing the letter $a$ divides every position containing the letter $b$. So if we interpret the sentence in strings over the alphabet $A = \{a, b, c\}$, we find that $aacbcbc$ satisfies the sentence, but $aacbbbc$ does not, because the latter string contains an $a$ in position 2 and a $b$ in position 5.

A sentence $\phi$ in this logic therefore defines a language in $A^*$, consisting of all words $w \in A^*$ such that $w \models \phi$. We denote by $FO[\mathcal{N}]$ the family of languages that can be defined in this way. The symbol $\mathcal{N}$ is meant to represent the class of all numerical predicates; the input alphabet $A$ is understood. We will often blur the distinction between sentences and the languages that they define, so we will sometimes write $w \models \phi$ as $w \in \phi$, and use $FO[\mathcal{N}]$ to denote the family of sentences as well as the associated family of languages.

This formalism for defining properties of words in first-order logic is connected on the one hand to boolean circuit complexity, and on the other to the theory of finite automata and regular languages. The connection to boolean circuit complexity is this: $FO[\mathcal{N}]$ is identical to the circuit complexity class $AC^0$ consisting of languages recognized by constant-depth polynomial-size boolean circuit families in which the $AND$- and $OR$- gates are permitted to have unbounded fan-in. This equivalence was shown by Immerman [7], and, independently, Gurevich and Lewis [5]. Low-depth circuit complexity is one of the very few parts of computational complexity theory for which we possess unconditional superpolynomial lower bounds: the principal result in this vein is the theorem of Furst, Saxe and Sipser [4] that PARITY—the set of bit strings in which the number of 1’s is even—is not in $AC^0$. More generally, one cannot determine in $AC^0$ whether the number of 1’s in a bit string is divisible by $k$, where $k$ is any positive integer greater than 1.

The connection with the theory of automata is this: If the numerical predicates used in the formulas are appropriately restricted, then the languages defined by sentences are regular languages. Indeed, much of the algebraic theory of regular languages begins with the theorem of Schützenberger [10], that if the only numerical predicate permitted is $<$, then we obtain precisely the regular languages whose syntactic monoids contain no nontrivial groups. We denote this class of languages by $FO[<]$.

We can add some other numerical predicates and still be able to define only regular languages. For example, we can write $x \equiv 0 \pmod{3}$ to say that a position is divisible by 3. In fact, if the only numerical predicates used can themselves be defined by finite automata (in a sense that we will make precise below), then the resulting sentences define regular languages. We can write this fact as

$$FO[\text{Reg}] \subseteq \text{Reg} \cap FO[\mathcal{N}].$$

2Strictly speaking, $AC^0$ is usually defined just strings over $\{0, 1\}$, while our logical formalism allows us to use any finite alphabet $A$ of input symbols. But this is not really a problem—one can, for example, adopt any fixed-length binary encoding of $A$ and thus recognize sets of strings over $A$ by circuits. The class of languages over $A$ obtained in this way does not depend on the encoding.

3This theorem is usually stated without reference to logic, using an equivalent formulation in which languages are defined by a kind of restricted regular expression (‘star-free’ languages). However the logical interpretation has been known for just as long.

2
Here we have used \textbf{Reg} to denote both the regular numerical predicates and the regular languages.

Barrington, et. al. [2], showed that the regular languages in $AC^0$ are precisely those in $FO[\text{Reg}]$, and thus

$$FO[\text{Reg}] = \text{Reg} \cap FO[\mathcal{N}].$$

This can be viewed as a reformulation of the theorem of Furst, Saxe and Sipser cited above: It is relatively easy to show, using either model-theoretic games or algebraic methods, that PARITY cannot be defined in $FO[\text{Reg}]$, so the equation above implies the circuit lower bound. On the other hand, the circuit complexity result was itself used to prove the equation, and we do not know of any other way to obtain what on the surface appears to be a result about logic and finite automata. It is an interesting problem to exploit this logic-automaton connection obtain a different kind of proof of the equation, one that does not use the circuit lower bounds. The present paper is intended as a contribution towards this goal.

### 1.2 The central conjecture, and the main result

Let $A$ be a finite alphabet, fixed throughout. $\Sigma_k[\mathcal{N}]$ denotes the class of languages in $A^*$ defined by $\Sigma_k$-sentences, in which arbitrary numerical predicates are permitted. $B\Sigma_k[\mathcal{N}]$ denotes the boolean closure of this class of languages. $B\Sigma_k[\text{Reg}]$ denotes the subclass in which only regular numerical predicates are permitted. (Precise definitions will be given below.)

**Conjecture 1** Let $k > 0$.

$$\Sigma_k[\mathcal{N}] \cap \text{Reg} = \Sigma_k[\text{Reg}]$$

and

$$B\Sigma_k[\mathcal{N}] \cap \text{Reg} = B\Sigma_k[\text{Reg}].$$

Observe that if the conjecture holds for $\Sigma_k$ sentences, then it holds as well for $\Pi_k$ sentences, by complementation.

This conjecture is explored at length in Straubing [11]. Our main result is that the conjecture is true when $k = 1$:

**Theorem 2**

$$\Sigma_1[\mathcal{N}] \cap \text{Reg} = \Sigma_1[\text{Reg}]$$

and

$$B\Sigma_1[\mathcal{N}] \cap \text{Reg} = B\Sigma_1[\text{Reg}].$$
1.3 Related work

Theorem 2 is not new: The equality for $\Sigma^1$ is quite easy and has been known for some time. Proofs appear in Péladeau [9] and Straubing [11]. We give the proof below, in Section 3, as it forms the basis for our subsequent argument. A proof of the equality for $B\Sigma^1$ is given by Maciel, et. al. [8]. A very different proof, which applies not just to boolean combinations of $\Sigma^1$ sentences but to the analogue using modular quantifiers, is given in Straubing [12]. What distinguishes our new proof from these earlier efforts is its very elementary nature. Both [8] and [12] used rather complex arguments, relying on circuit complexity, the algebraic characterization of dot-depth 1 languages and Ramsey theory. Instead, we show how to directly rewrite the defining formula for $L$ replacing the numerical predicates by regular numerical predicates, using only the most basic tools from set theory and automata theory.

Charles Paperman suggested to us that it was possible to find a proof along these lines, and we were able to work out the details once we understood the differencing trick in Section 4. The proof has already been presented in Borlido et. al., [3] as an example of an application of more general results on lattices and boolean algebras. We thought that the very abstract setting of that paper risks obscuring the relatively simple idea behind the proof, so we are posting this more self-contained presentation.

Recently, Barloy, et. al. [1], proved the central conjecture for $\Sigma^2$, with an intricate proof drawing on circuit complexity, extremal combinatorics, and algebra.

2 Notation

Following [11], we interpret sentences in strings over a finite alphabet $A$. More generally, we interpret formulas in which the free variables are contained in a set $V = \{x_1, \ldots, x_d\}$ of variables, in $V$-structures over $A$: These are words over the extended alphabet $A \times 2^V$ in which each variable appears exactly once. For example,

$$(a, \emptyset)(b, \{x_1, x_3\})(b, \emptyset)(a, \{x_2\})(a, \emptyset)$$

is a $\{x_1, x_2, x_3\}$-structure over $A = \{a, b\}$ that satisfies the formulas $x_1 = x_3$, $x_1 < x_2$, and $b(x_3)$, among others.

We use $\mathbf{v}$ to denote the $d$-tuple of variables $(x_1, \ldots, x_d)$. Thus if $\phi$ is quantifier-free, we write $\exists \mathbf{v} \phi$ for a $\Sigma^1$-formula. We similarly write $\mathbf{a}$ to denote a $d$-tuple of letters $(a_1, \ldots, a_d) \in A^d$. We write $\mathbf{a}(\mathbf{v})$ as an abbreviation for

$$\bigwedge_{i=1}^{d} a_i(x_i).$$

We deliberately overload notation: If $\phi$ is a sentence and $w \in A^*$, then we write $w \models \phi$ and $w \in \phi$ interchangeably, considering the logical formula and the language it defines.
as the same thing. Similarly, we consider sets of structures and the formulas with free variables that define these sets to be the same thing.

If \( i = (i_1, \ldots, i_d) \) is a \( d \)-tuple of positions in \( w \in A^* \), then we write \( w(i) \) to denote both the \( d \)-tuple of letters in these positions and the \( \{x_1, \ldots, x_d\} \) -structure obtained by tagging each position \( i_j \) with the corresponding variable \( x_j \). For example, the \( \{x_1, x_2, x_3\} \) -structure exhibited above would be denoted \( abbaa(2, 4, 2) \), but we also write \( abbaa(2, 4, 2) = (b, a, b) \).

A numerical predicate \( N \) with free variables \( V = \{x_1, \ldots, x_d\} \) is a set of \( V \)-structures with the following property: If \( w(i) \in N \), and \( v \) is obtained from \( w \) by changing one letter of \( w \), then \( v(i) \in N \). Thus satisfaction of the numerical predicate depends only on the length of the structure and the positions associated with the variables, and not on the letters in those positions. Thus we can write the satisfaction relation

\[
w(i) \models N(x)
\]

as

\[
(i; |w|) \models N(x).
\]

If we view the numerical predicate as a set of words over \( A \times 2^V \), then a regular numerical predicate is simply a numerical predicate that is a regular language over this alphabet. Regular numerical predicates are precisely those that can be expressed by first-order formulas over the base \( x_i < x_j \) and \( x_i \equiv 0 \pmod{q} \) for \( q > 0 \). (See [11] for a proof of this fact.)

### 3 The \( \Pi_1 \)-ceiling of a language, and the \( \Pi_1 \) case

Let \( L \subseteq A^*, d > 0 \). We will define a quantifier-free formula \( [L]_d \). The idea is that \( \forall x[L]_d \), where \( |x| = d \), will serve as a kind of closest approximation to \( L \) by a \( \Pi_1[N] \) sentence. We will call this the \( \Pi_1 \) ceiling of \( L \). (Obviously, this depends on \( d \).) However, in what follows, we will usually drop the subscript \( d \) - it will be understood that every quantifier block, tuple of indices, tuple of letters, etc., has size \( d \).

Given \( a \in A^d \), we define a numerical predicate \( R^L_a(x) \) by

\[
(i; n) \models R^L_a(x)
\]

if and only if there is some \( v \in L \) with \( v(i) = a \) and \( |v| = n \). We define \( [L] \) to be the quantifier-free formula

\[
\bigvee_{a \in A^d} (a(x) \land R^L_a(x)).
\]

The next Proposition gives some basic properties of the ceiling operator.

**Proposition 3**

1. \( L \subseteq \forall x[L] \).
2. If $L$ is itself defined by a $\Pi_1$ sentence, then $L = \forall x[L]$.

3. If $L_1 \subseteq L_2$, then $[L_1] \subseteq [L_2]$.

4. If $L$ is a regular language, then $[L]$ uses only regular numerical predicates.

**Proof**

1. Let $w \in L$. Let $i$ be a $d$-tuple of positions in $w$. Then $w(i) \models a(x)$ for exactly one $a \in A^d$. Since $w \in L$, by definition $w(i) \models R^L_a(x)$. Since $i$ was arbitrary, we have $w \models \forall x[L]$.

2. Now suppose $L = \forall x\phi$ for some quantifier-free $\phi$. We can write a formula equivalent to $\phi$ in disjunctive normal form as

$$\bigvee_{a \in A^d} (a(x) \land N_a(x)),$$

where each $N_a$ is a numerical predicate. We claim $L = \forall x[L]$. We already have inclusion from left to right. So now suppose $w \in \forall x[L]$, and suppose, contrary to what we want to prove, that $w \notin L$. Thus there is some $d$-tuple $i$ of positions in $w$ such that $w(i) \not\models \phi$. Let $a = w(i)$. Then $w(i) \not\models N_a(x)$. However, by assumption, $w(i) \models R^L_a(x)$. Thus there is some $v \in L$ with $v(i) = a$ and $|v| = |w|$. Since $v \in L$ we must then have $v(i) \models N(x)$. But since $N_a$ is a numerical predicate, we would then also have $w(i) \models N_a(x)$, a contradiction.

3. Clearly if $L_1 \subseteq L_2$ then $R^{L_1}_a(x) \subseteq R^{L_2}_a(x)$, from which the result follows.

4. We have to show that if $L$ is a regular language and $a \in A^d$, then there is a finite automaton $A$ over the alphabet $A \times 2^{\{x_1, \ldots, x_d\}}$ such that the set of $\{x_1, \ldots, x_d\}$-structures accepted by $A$ is exactly $R^L_a(x)$. To construct $A$, begin with an automaton that recognizes $L$. Then replace every edge

$$q \xrightarrow{a} q',$$

where $a \in A$, by all the edges

$$q \xrightarrow{(b, S)} q',$$

where $b \in A$, and $S \subseteq \{x_j : a_j = a\}$.

That is, we replace a single edge labeled $a$ in the original automaton by multiple edges. These can be labeled by *any* letter of $A$. The variables contained in the second component $S$ must correspond to components of the tuple $a$ that contain the letter $a$. The new automaton $A$ is non-deterministic.

Keep the same initial and accepting states as in the original automaton that recognizes $L$. The set of structures accepted by this new automaton $A$ is $R^L_a(x)$. The reason is that each accepting path in $A$ labeled by a structure $v(i)$ traverses the same states as an accepting path in the original automaton labeled by a word $w$ with $w(i) = a$. Our new automaton might accept words over $A \times 2^{\{x_1, \ldots, x_d\}}$ that are not structures, because they do not contain
each variable exactly once. However, the set of legal structures is itself a regular language, so we get regularity of $R^L_a(x)$ by intersection. Easily, the set of structures $a(x)$ is also regular, and it follows that $\lceil L \rceil$ is regular. □

Corollary 4

$$\Pi_1[\mathcal{N}] \cap \text{Reg} = \Pi_1[\text{Reg}].$$

$$\Sigma_1[\mathcal{N}] \cap \text{Reg} = \Sigma_1[\text{Reg}].$$

Proof

If a regular language $L$ is defined by a $\Pi_1$ sentence, then property 2 of Proposition 3 implies that $L = \forall x \lceil L \rceil$, and property 4 implies that this uses only regular numerical predicates. If a regular language $L$ is defined by a $\Sigma_1$ sentence $\psi$, then its complement is defined by $\lnot \psi$, which is equivalent to a $\Pi_1$ sentence, and the result follows from the property for $\Pi_1$. □

4 Iterated differences of monotone formulas

In order to extend Corollary 4 to arbitrary boolean combinations of $\Pi_1$ sentences, we will employ a kind of normal form for propositional logic, expressing every propositional formula as an iterated difference of monotone formulas. Since the existence of this normal form is not as obvious or as well-known as other representations of propositional formulas, we will give a complete proof. (The use of such chains of differences goes back to Hausdorff [5]—see the discussion in Borlido, et al. [3].)

We consider propositional formulas over a finite set of variables $\{p, q, r, \ldots\}$. A formula is monotone if in every satisfying assignment, changing the value of one variable from False to True yields another satisfying assignment. A formula is monotone if and only if it is equivalent to a formula in which the negation symbol does not appear (this includes the formulas T and F). Normal-form formulas are defined recursively as follows:

1. Every monotone formula is in normal form.

2. If $\phi$ is monotone and $\psi$ is in normal form, then $\phi \land \lnot \psi$ is in normal form.

We will also write $\phi \land \lnot \psi$ as $\phi - \psi$, so a formula in normal form looks like

$$\phi_1 - (\phi_2 - (\phi_3 \ldots \cdots)).$$

Proposition 5 Every propositional formula is equivalent to a formula in normal form.

Proof
We will identify propositional formulas with \( k \) variables by sets of bit strings of length \( k \) representing the satisfying assignments for the formula. So, for example,
\[
(p \land \neg q) \lor (r \land \neg s)
\]
is identified with
\[
\{1000, 1001, 1010, 1011, 0010, 0110, 1110\}.
\]
In particular, we treat a formula as identical to its set of satisfying assignments, and so do not distinguish between equivalent formulas.

There is the obvious partial order on bit strings: \( a \leq b \) if \( b \) is obtained from \( a \) by switching a (possibly empty) set of 0’s to 1’s. Easily, monotone propositional formulas are those corresponding to sets of bit strings that are upward closed for this partial order. If \( \phi \) is a propositional formula, then \( \uparrow \phi \) represents the upward closure of \( \phi \), which is then a monotone formula. Considered as a set of bit strings, \( \uparrow \phi \) is the intersection of all the monotone formulas that contain \( \phi \).

We claim that the following algorithm computes the normal form: Given a set \( X \) of bit strings, we write
\[
X = \uparrow X - (\uparrow X - X).
\]
If \( \uparrow X = X \), that is, if \( X \) is already monotone, then \( X \) is already in normal form, and we can leave off the second term. In particular, if \( X = \emptyset \), then \( X \) is monotone, so there is nothing to do. Otherwise, we apply the algorithm recursively to \( X' = \uparrow X - X \), until we reach a monotone result. We need to prove that the algorithm terminates, which follows from the claim:

**Claim 6** If \( X \neq \emptyset \), then
\[
\uparrow (\uparrow X - X) \subsetneq \uparrow X.
\]
This ensures that the sequence of monotone formulas generated by the algorithm strictly decreases in cardinality, so we will eventually get to a place where \( \uparrow X - X \) is empty.

To prove the claim, observe that \( \uparrow \) is inclusion-preserving: that is, for any sets \( U, V \) of bit strings, if \( U \subseteq V \), then \( \uparrow U \subseteq \uparrow V \). Thus, in particular
\[
\uparrow (\uparrow X - X) \subseteq \uparrow \uparrow X = \uparrow X.
\]
To see that the inclusion is strict, we use the hypothesis that \( X \) is nonempty and pick a minimal element \( x \) of \( X \). Then \( \uparrow X - \{x\} \) is monotone. Combining this with the order-preserving property, we get
\[
\uparrow (\uparrow X - X) \subseteq \uparrow (\uparrow X - \{x\}) = \uparrow X - \{x\} \subseteq \uparrow X,
\]
proving the claim. □
Example. Let \( k = 3 \) and let \( \phi \) have satisfying assignments \( 101, 010, 111 \). We can write this formula in disjunctive normal form as

\[
(p \land \neg q \land r) \lor (\neg p \land q \land \neg r) \lor (p \land q \land r).
\]

We have

\[
\uparrow \phi = \{(0,1,0),(1,1,0),(0,1,1),(1,0,1),(1,1,1)\} = q \lor (p \land r).
\]

\[
\phi' = \uparrow \phi - \phi = \{(1,1,0),(0,1,1)\}.
\]

\[
\uparrow \phi' = \{(1,1,0),(0,1,1),(1,1,1)\} = q \land (p \lor r).
\]

\[
\uparrow \phi' - \phi' = \{(1,1,1)\} = p \land q \land r.
\]

This last formula is itself monotone, so the algorithm terminates here. Thus \( \phi \) is equivalent to the normal form formula

\[
(q \land (p \lor r)) - ((q \lor (p \land r)) - (p \land q \land r)).
\]

5 Proof of the main theorem

A boolean combination of \( \Sigma_1 \) sentences is equivalent to a propositional formula in which the atoms are replaced by sentences of the form \( \forall x \phi \), where \( \phi \) is quantifier-free. Now observe that a monotone formula in these atoms can be replaced by a single atom, because the class of \( \Pi_1 \) formulas is closed under both conjunction and disjunction. Note that for disjunction we need more variables—that is,

\[
\forall x \phi_1(x) \lor \forall x \psi(x) \equiv \forall xx' (\phi(x) \lor \psi(x')),
\]

where \( x' \) is a new copy of the variables. With conjunction you don’t need to increase the number of variables.

Thus by our normal form theorem, every sentence \( B\Sigma_1[\forall] \) is equivalent to an iterated difference

\[
\forall x \phi_1 - (\forall x \phi_2 - (\forall x \phi_3 - \cdots (\forall x \phi_k) \cdots)).
\]

We can insure that all of the blocks of variables \( x \) in this formula have the same length \( d \) by adding additional dummy variables.

Let \( L \subseteq A^* \). We define \( L \)-derived languages and \( L \)-derived quantifier-free formulas recursively as follows:

1. \( L \) is an \( L \)-derived language.

2. Any boolean combination of \( L \)-derived languages is an \( L \)-derived language.

3. If \( \theta \) is an \( L \)-derived quantifier-free formula with free variables \( x \), then \( \forall x \theta \) is an \( L \)-derived language.
4. If $K$ is an $L$-derived language and $d > 0$, then $\lceil K \rceil_d$ is an $L$-derived quantifier-free formula.

Now suppose $L$ is a language in $B\Sigma_1[N]$, so that

$$L = \forall x \phi_1 - (\forall x \phi_2 - (\forall x \phi_3 - \cdots (\forall x \phi_k)\cdots)),$$

as above. We claim that $L$ is defined by a sentence

$$\forall x \psi_1 - (\forall x \psi_2 - (\forall x \psi_3 - \cdots (\forall x \psi_k)\cdots)),$$

where each $\psi_i$ is an $L$-derived quantifier-free formula. This gives the result about $B\Sigma_1[N] \cap \text{Reg}$ as a special case, since we showed earlier that when $L$ is regular, every $L$-derived quantifier-free formula uses only regular numerical predicates.

We proceed by induction on $k$, and note that the case $k = 1$ was done in Section 3.

Now suppose $k > 1$. Let

$$L' = \forall x [L].$$

Further, let

$$K_1 = \forall x \phi_1, K_2 = \forall x \phi_2 - (\forall x \phi_3 - \cdots (\forall x \phi_k)\cdots)$$

so that $L = K_1 - K_2$. Since $L \subseteq K_1$, monotonicity shows that $L' \subseteq \forall x [K_1]$, and since $K_1$ is defined by a $\Pi_1$-sentence, this last is equal to $K_1$ (by Property 2 of the ceiling operator). So $L' \subseteq K_1$. We thus have

$$L' - L = L' \cap \overline{L} = L' \cap K_1 \cap K_2 = L' \cap (K_1 \cup K_2) = (L' - K_1) \cup (L' \cap K_2) = L' \cap K_2,$$

the last line following from our observation that $L' \subseteq K_1$.

$L'$ is defined by a $\Pi_1$-sentence, and $K_2$ is defined by an iterated difference with $k - 1$ $\Pi_1$ sentences as terms. It follows that their intersection $L' \cap K_2$ is defined by an iterated difference with $k - 1$ terms. This is because of the identity

$$p \land (q - r) \equiv (p \land q) - r,$$

so that if $p$ and $q$ are both $\Pi_1$ sentences, then, $p \land q$ is as well. Thus $L' - L$ is defined by an iterated difference with $k - 1$ terms. By the inductive hypothesis, $L' - L$ is defined by such a difference $\theta$ in which every quantifier-free formula is $(L' - L)$-derived, hence $L$-derived. Thus, since $L \subseteq L'$,

$$L = L' - (L' - L) = \forall x [L] - \theta,$$

which is an iterated difference with $k$ terms, each of which uses only $L$-derived quantifier-free formulas.
6 Further work

We would like to extend this argument to prove the conjecture for values of \( k \) greater than 1. One obstacle to doing this is figuring out exactly how to generalize the ceiling operator. Here is one idea that one might try: Any \( \Pi_{k+1} \) sentence, where \( k \geq 1 \), has the form

\[
\forall x \phi,
\]

where \( \phi \) is a boolean combination of \( \Sigma_k \) formulas whose free variables are contained in \( V = \{x_1, \ldots, x_d\} \). While \( \Sigma_k \) formulas in general can have arbitrarily large quantifier depth and use arbitrarily many numerical predicates, a fixed formula contains only finitely many numerical predicates, and the \( \Sigma_k \) formulas have bounded quantifier depth. Thus we will restrict our attention to formulas \( \phi \) that are boolean combinations of \( \Sigma_k \) formulas using numerical predicates from a fixed finite set, and whose quantifier depth is bounded above by some constant \( t \). We will call these \textit{conforming} formulas: For any choice of parameters \( d, t \) and a finite set of numerical predicates, there are only finitely many inequivalent conforming formulas. Now suppose \( L \subseteq A^* \), and consider the set of \( V \)-structures

\[
\hat{L} = \bigvee_{u \in L, j \in |u|^d} F(u, j),
\]

where \( F(u, j) \) is the conjunction of all the conforming formulas satisfied by \( u(j) \). Since this conjunction is finite, \( F(u, j) \) is itself a conforming formula. And while the disjunction is over infinitely many structures \( u(j) \), there are only finitely many distinct disjuncts, so \( \hat{L} \) itself is a conforming formula. It is easy to show that \( \hat{L} \) has the first three properties of our ceiling operator \( \lceil L \rceil \), enumerated in Proposition 3. However, we see no way to show that it has the crucial fourth property of preserving regularity. If it did, it would be possible to extend our argument by induction to higher levels, and prove Conjecture 1.

Here is a problem that may be simpler to solve: Suppose that the first part of the conjecture

\[
\Sigma_k[N] \cap \text{Reg} = \Sigma_k[\text{Reg}]
\]

holds for some \( k \). Can we then use the methods of this paper to extend this to the boolean closure \( B\Sigma_k \)? As we mentioned earlier, there is a new proof of the conjecture for \( \Sigma_2 \), so such a stratagem would settle the conjecture for all sentences with a single quantifier alternation.

One of the goals of this research is to find a different kind of proof of the theorem of Furst, Saxe and Sipser discussed in the introduction, by recasting it as a problem about automata and logic and attacking this translation of the problem directly. This is not a terribly ambitious goal, since, after all, the theorem has already been proved in a number of different ways, the proofs are not exceptionally difficult, and actually give stronger lower
bounds than would be obtained by what we propose here. However, Barrington et. al. [2] also discuss the analogous identity when one allows both ordinary and modular quantifiers:

$$FOMOD[\text{Reg}] \subseteq \text{Reg} \cap FOMOD[\mathbb{N}],$$

and show that this is equivalent to the long-open conjecture in circuit complexity that the class $ACC^0$ is strictly smaller than $NC^1$. Thus an approach that works directly on these reformulated versions of the circuit problems would be of considerable interest.

References

[1] Corentin Barloy, Michaël Cadilhac, Charles Paperman, and Thomas Zeume. The regular languages of first-order logic with one alternation. arXiv preprint arXiv:2203.06075, 2022.

[2] David A Mix Barrington, Kevin Compton, Howard Straubing, and Denis Thérien. Regular languages in $nc1$. Journal of Computer and System Sciences, 44(3):478–499, 1992.

[3] Célia Borlido, Mai Gehrke, Andreas Krebs, and Howard Straubing. Difference hierarchies and duality with an application to formal languages. Topology and its Applications, 273:106975, 2020.

[4] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. Mathematical systems theory, 17(1):13–27, 1984.

[5] Yuri Gurevich and Harry R. Lewis. A logic for constant-depth circuits. Information and Control, 61(1):65–74, 1984.

[6] Felix Hausdorff. Set theory, volume 119. American Mathematical Soc., 2005.

[7] Neil Immerman. Languages that capture complexity classes. SIAM Journal on Computing, 16(4):760–778, 1987.

[8] Alexis Maciel, Pierre Péladeau, and Denis Thérien. Programs over semigroups of dot-depth one. Theoretical Computer Science, 245(1):135–148, 2000.

[9] Pierre Péladeau. Classes de circuits booléens et variétés de monoïdes. PhD thesis, Paris 6, 1990.

[10] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. Inf. Control., 8(2):190–194, 1965.

[11] Howard Straubing. Finite automata, formal logic, and circuit complexity. Birkhäuser, Boston, 1994.
[12] Howard Straubing. Languages defined with modular counting quantifiers. *Information and Computation*, 166(2):112–132, 2001.