A BRIEF SURVEY OF INDUCTIVE DESCRIPTIVE COMPLEXITY

WILL ASNESS

Abstract. First order logic is not very powerful on its own—but what if we were allowed to analyze the result of repeated application of a first order formula? In this paper, we demonstrate that, by adding the power of induction to first-order logic, one is able to capture some important complexity classes by modifying what, exactly, we mean by "induction." In doing so, we provide direct proofs that P = FO(LFP), PSPACE = FO(PFP), and NL = FO(pos-TC).

Contents

1. Introduction

In complexity theory, we find that the standard measures of complexity of a computation, space and time, are very natural in terms of their practical origin—who doesn't want to know how much space they need to allocate or how much time they need to wait in order to compute a query? It may seem on the face rather difficult to connect the description of a problem with a lower bound on the resources it requires (i.e. intrinsic difficulty to solve), since, from a purely mathematical point of view, the measurements of space and time are somewhat arbitrary. Thus, descriptive complexity aims to characterize complexity classes through the richness of the language required to define a query in the complexity class, which more naturally intertwines the definition of a problem with the resources needed to compute it.

One very simple form of statement in mathematical logic is the "first order statement," which allows quantification over only objects within a structure (as opposed to subsets of or relations on the structure). These statements are not very powerful, but they can be computed very quickly; thus, they can be seen as a building block, out of which we can construct more complex queries. In this paper, we demonstrate some key theorems that have been direct results of this line of thought and provide intuition as to why, exactly, the forms of induction presented capture the complexity classes we discuss.

In order to understand this paper fully, the reader should have a basic understanding of complexity theory.

2. Background in Logic

Before we begin our quest for Descriptive Complexity, we clarify some terminology. In order to utilize a logical language, we must first define what our languages consist of.

Definition 2.1. A *vocabulary* is a tuple of relation symbols and constant symbols τ, defined by

$$\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle.$$

For each i, the symbol $R_i^{a_i}$ denotes a relation symbol of arity a_i.

Note that, traditionally, vocabularies are defined including function symbols and, indeed, a vocabulary as we defined it above is often referred to as a *relational vocabulary.* This does not restrict us in our current analysis; we can encode function symbols of arity a into a relation symbol of arity $a + 1$ and, if one looks at the definition of $\text{bin}_\tau(\mathcal{A})$ below, the way that a function symbol would intuitively be encoded is, indeed, the same as encoding it as a tuple.

Given a vocabulary τ, we would like to have objects we can discuss using this vocabulary. This gives rise to the following definition:

Definition 2.2. A *structure* with a vocabulary $\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$ is 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 a non-empty set (referred to as the "universe" of $\mathcal{A}$), for each $R_i^{a_i} \in \tau$, we have defined $R_i^{\mathcal{A}} \subset |\mathcal{A}|^{a_i}$ (i.e., $R_i^{\mathcal{A}}$ is an a_i-ary relation on $|\mathcal{A}|$), and for each $c_i \in \tau$, we have some specified element $c_i^{\mathcal{A}} \in |\mathcal{A}|$. Further, let $||\mathcal{A}||$ denote the cardinality of $|\mathcal{A}|$.

Since we are dealing with computation in this paper, we only care about finite structures; thus, we let STRUC$[\tau]$ be the set of all finite structures of vocabulary τ. The definition of a structure is a bit difficult to parse, and possibly non-intuitive, and so we provide two nice examples.

Example 2.3 (Vocabulary of Graphs)**.** The vocabulary of graphs with a specified start and sink node is $\tau_g = \langle E^2, s, t \rangle$, in which E is a binary relation specifying which edges are present in the graph, s is the start node, and t is a sink node. A structure in τ_g is

$$G = \langle V^G, E^G, 1, 5 \rangle$$

where

$$V^G = \{1, 2, 3, 4, 5\}, \qquad E^G = \{(1, 2), (1, 4), (3, 5), (4, 3), (4, 5)\}.$$

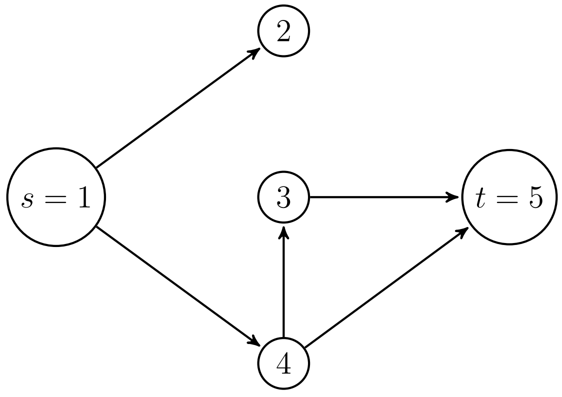


FIGURE 1. The Graph G from Example 2.3

Example 2.4 (Cyclic Group)**.** A vocabulary for a group is $\tau_G = \langle \cdot^3, \mathbf{1} \rangle$ (we will see later how to write the group axioms in an easy-to-evaluate way in order to ensure that a given structure is a group). To define the cyclic group of order 3, we define

$$C = \langle V^C, \cdot^C, e \rangle$$

where

$$\begin{aligned} V^C &= \{e, z^1, z^2\} \\ \cdot^C &= \{(e,e,e), (e,z^1,z^1), (e,z^2,z^2), (z^1,e,z^1), (z^1,z^1,z^2), \\ &\quad (z^1,z^2,e), (z^2,e,z^2), (z^2,z^1,e), (z^2,z^2,z^1)\}. \end{aligned}$$

Under inspection, this corresponds to the multiplication table,

$\cdot^G$	e	z^1	z^2
e	e	z^1	z^2
z^1	z^1	z^2	e
z^2	z^2	e	z^1

and thus represents the object we desire.

Now, given vocabularies and objects of those vocabularies, we want to be able to write well-defined sentences about these objects in these vocabularies.

Definition 2.5. Given a vocabulary τ, the *first-order language* $\mathcal{L}(\tau)$ is the set of formulas built up from relation symbols of τ, constant symbols of τ, the logical relation symbol $=$, the connectives $\wedge$ and $\neg$, a set of variables VAR $= \{x_1, x_2, \cdots\}$, and the quantifier $\exists$. The methodology for building sentences is defined recursively:

(i) If a and b are constants or variables, then $a = b$ is a formula
(ii) If R is a k-ary relation in τ and $x_1, \cdots, x_k$ are constants/variables, then $R(x_1, \cdots, x_k)$ is a formula.
(iii) If ϕ, ψ are formulas, then $\phi \wedge \psi$ and $\neg\phi$ are formulas.
(iv) If x is a variable symbol and ϕ is a formula, then $(\exists x)\phi$ is a formula.

Given a formula, such as the formula in τ_g

$$(\exists z)(E(s,z) \wedge E(z,t)), \tag{2.6}$$

we want to evaluate whether or not it is "true" in a given structure. First, we must denote an instance of a variable v in a formula to be *bound* if it lies in the scope of a quantifier, $(\exists v)$, and otherwise it is *free.* Note that bound variables can be renamed in a formula without changing the formula, but free variables cannot. Now, let an *interpretation* into a τ-structure $\mathcal{A}$ be a function $i\colon V \to |\mathcal{A}|$, where V is a finite subset of VAR. We suppose that, for any constant symbol $c \in \tau$, we have that $c \in V$ and $i(c) = c^{\mathcal{A}}$. Using interpretations, we can define the "truth" of a statement in a purely formal way.

Definition 2.7 (Truth of a formula)**.** Let $A \in \mathrm{STRUC}[\tau]$ and let i be an interpretation into $\mathcal{A}$ which is defined on all relevant free variables ("relevance" to be determined by the recurrence below). Then, for all $\varphi \in \mathcal{L}(\tau)$, we say that $(\mathcal{A}, i) \vDash \varphi$ if the corresponding condition below holds (using $\Leftrightarrow$ to signify "if and only if")

$$\begin{aligned}
(\mathcal{A}, i) \vDash t_1 = t_2 &\Leftrightarrow i(t_1) = i(t_2) \\
(\mathcal{A}, i) \vDash R_j(t_1, \ldots, t_{a_j}) &\Leftrightarrow \langle i(t_1), \ldots, i(t_{a_j})\rangle \in R_j^{\mathcal{A}} \\
(\mathcal{A}, i) \vDash \neg\varphi &\Leftrightarrow \text{ it is not true that } (\mathcal{A}, i) \vDash \varphi \\
(\mathcal{A}, i) \vDash \varphi \wedge \psi &\Leftrightarrow (\mathcal{A}, i) \vDash \varphi \text{ and } (\mathcal{A}, i) \vDash \psi \\
(\mathcal{A}, i) \vDash (\exists x)\varphi &\Leftrightarrow \text{there exists } a \in |\mathcal{A}| \text{ such that } (\mathcal{A}, i_{x,a}) \vDash \varphi, \\
&\qquad \text{where } i_{x,a}(y) = \begin{cases} i(y) & \text{if } y \neq x \\ a & \text{if } y = x \end{cases}.
\end{aligned}$$

We write $\mathcal{A} \vDash \varphi$ if $(\mathcal{A}, i) \vDash \varphi$ where i is the interpretation only defined on constants.

To demonstrate the equality of two sentences, we use the symbol $\equiv$. Now, we define the abbreviated quantifiers "for all" and the connectives "or," "implies," and "propositional equivalence":

$$\begin{aligned}
(\forall x)\varphi &\equiv \neg(\exists x)\neg\varphi; \\
\alpha \vee \beta &\equiv \neg(\neg\alpha \wedge \neg\beta); \\
\alpha \to \beta &\equiv \neg\alpha \vee \beta; \\
\alpha \leftrightarrow \beta &\equiv (\alpha \to \beta) \wedge (\beta \to \alpha)\,.
\end{aligned}$$

note, in the above definitions, we assign $\neg$ the highest precedence, then $=$, then $\wedge$ and $\vee$, then $\to$ and $\leftrightarrow$, and finally quantifiers $\exists$ and $\forall$.

Finally, we have enough tools to construct nice sentences about the structures we built above! Below are a few examples.

Example 2.8. Recall τ_g from Example 2.3, the vocabulary of graphs. If we wanted to express that $G \in \mathrm{STRUC}[\tau_g]$ is *undirected*, e.g. the edge relation is symmetric, then we only have to check if $G \vDash \phi_{\mathrm{undir}}$, where

$$\phi_{\mathrm{undir}} \equiv (\forall xy)(E(x, y) \to E(y, x)).$$

Here, $\forall xy$ is an abbreviation for $\forall x \forall y$, and we have a similar abbreviation for $\exists xy \equiv \exists x \exists y$.

Similarly, we can ask if G is regular of out-degree 1, which means that, for each node x, there is exactly 1 distinct node, which is not x, which has an edge going into it from x. Note this implies that G has no self-edges. Formally, we can see $G \vDash \phi_{r1}$, where

$$\phi_{r1} \equiv (\forall x\ \neg E(x,x)) \wedge \forall x \exists y (E(x,y) \wedge \forall z (E(x,z) \to z = y))$$

Example 2.9. Now, recall τ_G from Example 2.4. We make τ_G even less structured by working with $\tau_{bin} = \langle \cdot^3 \rangle$, which is simply the vocabulary of a binary operation. We can check whether a structure $G \in \mathrm{STRUC}[\tau_{bin}]$ satisfies the group axioms using these first-order statements. First, we must make sure that $\cdot$ is well defined, as in, for all pairs of elements, there is a unique element that the pair maps to.

$$\phi_{def} \equiv \forall ab \exists z (\cdot(a,b,z) \wedge \forall y (\cdot(a,b,y) \to y = z)).$$

We then check the associativity holds, i.e. $G \vDash \phi_{\text{associativity}}$, where

$$\phi_{\text{associativity}} \equiv \forall xyz\ \exists fgh\ \cdot(x,y,f) \wedge \cdot(f,z,h) \wedge \cdot(y,z,g) \wedge \cdot(x,g,h).$$

Since our vocabulary is relational, we must use dummy variables in order access the "results" of the "operation" $\cdot^3$. Thus, we essentially have to say, "given elements x, y, z, there exist elements f, g, h such that $x \cdot y = f$ and $f \cdot z = h$, and also $y \cdot z = g$ and $x \cdot g = h$." The other two axioms, identity and existence of an inverse, are simpler. However, we start with a "helper" statement that takes a free variable, e, and checks whether it is an identity. Then, existence of an identity can be checked with $G \vDash (\exists i)\phi_{\text{id}}(i)$, where

$$\phi_{\text{id}}(e) \equiv \forall x\ \cdot(e,x,x) \wedge \cdot(x,e,x).$$

The shorthand $\phi_{\text{id}}(i)$ represents formula ϕ_{id} where we see e as representing the same element as i. Then, we can use this to check for existence of an inverse:

$$\phi_{\text{inverse}} \equiv \exists e (\phi_{\text{id}}(e) \wedge \forall a \exists b \cdot(a,b,e) \wedge \cdot(b,a,e)).$$

There is a trend where we want to ask for existence of a *unique* element; we do so with $\exists!$, or "there exists a unique," which can be defined by

$$\exists! z\ \phi(z) \equiv \exists z (\phi(z) \wedge \forall y (\phi(y) \to y = z)).$$

For example, above, we could have defined above

$$\phi_{def} \equiv \forall ab\ \exists! z\ \cdot(a,b,z).$$

Finally, as a general method for mapping between structures, we define:

Definition 2.10. A *query* is any mapping $I \colon \mathrm{STRUC}[\sigma] \to \mathrm{STRUC}[\tau]$, where σ and τ are arbitrary vocabularies. In particular, a *boolean query* is a mapping $I \colon \mathrm{STRUC}[\sigma] \to \{0,1\}$.

Now that we have a formal language to talk about these structures, map between them, and ask boolean questions about them, we turn to complexity theory to look at the complexity of computing properties of these structures.

3. Background in Complexity

In Computational Complexity, the standard model for capturing the abstract concept of "computation" is the *Turing Machine*, and that is the model we use here. We assume the reader is familiar with the construction of a Turing Machine, and can refer to [Aro09] for a very nice definition.

For a Turing machine (abbreviated TM) M and a binary string w, we write $M(w)\downarrow$ to say that M accepts the input w, and we define

$$L(M) = \{w \in \{0,1\}^* \colon M(w)\downarrow\},$$

and so $L(M)$ is the language accepted by M. We use in this definition that, for any set K, we define $K^* = \bigcup_{n\in\mathbb{N}} K^n$ (as in, the set of finite strings in K) and that a language is any $L \subset \{0,1\}^*$. Now, in order to compute queries on the finite structures defined above, we want some way to convert a finite structure into a binary string that is polynomial in length with respect to the number of elements in the structure.

Definition 3.1. Let $\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s\rangle$ be a vocabulary, and let $\mathcal{A} = \langle, \{0, 1, \ldots, n-1\}, R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s\rangle$. We define $\text{bin}_\tau \colon \text{STRUC}[\tau] \to \text{STRUC}[\tau_s]$, for $\tau_s = \langle S^1, \leq^2\rangle$ to be the binary encoding function for the language τ. Let $\mathcal{A} \in \text{STRUC}[\tau]$ such that $|\mathcal{A}| = \{0, 1, \ldots, n-1\}$. Then, for the a_i-ary relation R_i in τ, we define $\text{bin}^{\mathcal{A}}(R_i)$ to be the n^{a_i}-length binary string such that index $b_0 \cdot n^0 + \ldots b_{a_i-1} \cdot n^{a_i-1}$ is a 1 if and only if $R_i(b_0, \ldots, b_{a_i-1})$ holds in $\mathcal{A}$. Similarly, we can encode a constant c_i in $\lceil \log n\rceil$ bits, represented as $\text{bin}^{\mathcal{A}}(c_i)$. Thus, we define $\text{bin}_\tau(\mathcal{A})$ to be the concatenation of these, represented as

$$\text{bin}_\tau(\mathcal{A}) = \text{bin}^{\mathcal{A}}(R_1)\text{bin}^{\mathcal{A}}(R_2)\cdots\text{bin}^{\mathcal{A}}(R_r)\text{bin}^{\mathcal{A}}(c_1)\cdots\text{bin}^{\mathcal{A}}(c_s),$$

and so

$$||\text{bin}_\tau(\mathcal{A})|| = n^{a_1} + \cdots + n^{a_r} + s\lceil \log n\rceil.$$

This definition intrinsically necessitates an ordering on the universe of τ in order for this definition to make sense. That is why, for the sake of this paper, we assume that all structures have a total ordering $\leq$ on them, and a definition for BIT^3 that allows us to look at elements of a structure as binary numbers, where $\text{BIT}(x, y, k)$ means that the yth bit of x has the value k.

It is important to observe that this definition is a first-order reduction, which is defined below. Furthermore, since, for a given structure $\mathcal{A}$ the vocabulary is implied from its construction, we often write $\text{bin}(\mathcal{A})$ rather than $\text{bin}_\tau(\mathcal{A})$ for simplicity. Given that we now have a binary encoding of models, we can perform computations on them.

Definition 3.2. Let $I \colon \text{STRUC}[\sigma] \to \text{STRUC}[\tau]$ be a query. Let T be a Turing machine. Suppose that for all $\mathcal{A} \in \text{STRUC}[\sigma]$, we have that $T(\text{bin}(\mathcal{A})) = \text{bin}(I(\mathcal{A}))$; in this case, we say that T *computes* I.

In particular, if I is a boolean query, then T accepts the input if and only if I holds true. Then, we define $\text{DTIME}[t(n)]$ to be the set of boolean queries I computable by a Turing machine T that takes $O(t(n))$ steps (where n is the size of the universe of the input structure). We similarly define $\text{NTIME}[t(n)]$, $\text{DSPACE}[s(n)]$, and $\text{NSPACE}[s(n)]$ for nondeterministic time, deterministic space, and nondeterministic space respectively. Now, we can define

$$\begin{aligned} \mathrm{L} &= \mathrm{DSPACE}[\log n], \\ \mathrm{NL} &= \mathrm{NSPACE}[\log n], \\ \mathrm{P} &= \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}[n^k], \quad \text{and} \\ \mathrm{PSPACE} &= \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}[n^k], \end{aligned}$$

which will be four of the main complexity classes we utilize in this paper. Observe that

$$\mathrm{L} \subset \mathrm{NL} \subset \mathrm{P} \subset \mathrm{PSPACE},$$

with NL $\subsetneq$ PSPACE (this can be proved using the fact that $\mathrm{NSPACE}[s(n)] \subset \mathrm{DSPACE}[s(n)^2]$ and space hierarchy theorem). We also define FO to be the complexity class of first-order queries; that is, the set of queries $I\colon \mathrm{STRUC}[\tau] \to \{0,1\}$ such that there exists some $\varphi \in \mathcal{L}(\tau)$ where $I(\mathcal{A})$ holds if and only if φ holds in $\mathcal{A}$.

Each boolean query $I\colon \mathrm{STRUC}[\tau] \to \{0,1\}$ can be seen as a subset of $\mathrm{STRUC}[\tau]$ or $\{0,1\}^*$, where the corresponding subset to a query I would be $\{\mathcal{A}\colon I(\mathcal{A})\}$ or $\{\mathrm{bin}(\mathcal{A})\colon I(\mathcal{A})\}$, respectively. With this in mind, we can define a *complexity class* to be any set of subsets of $\{0,1\}^*$ (by this definition, $\mathrm{FO}, \mathrm{L}, \mathrm{NL}, \mathrm{P},$ and PSPACE are all complexity classes). This brings us to the next, general definition, which we will use to connect complexity and logic.

Definition 3.3. Let $I\colon \mathrm{STRUC}[\sigma] \to \mathrm{STRUC}[\tau]$ be a query. We say that I is *computable in* $\mathcal{C}$ if and only if the boolean query I_b is an element of $\mathcal{C}$, where

$$I_b = \{(\mathcal{A}, i, a) \mid \text{ the } i^{\text{th}} \text{ bit of } \mathrm{bin}(I(\mathcal{A})) \text{ is ``a''}\}.$$

Let $\mathcal{Q}(\mathcal{C})$ be the set of all queries computable in $\mathcal{C}$:

$$\mathcal{Q}(\mathcal{C}) = \mathcal{C} \cup \{I \mid I_b \in \mathcal{C}\}.$$

This notation is useful in the following definition:

Definition 3.4 (Many-One Reduction)**.** Let $\mathcal{C}$ be a complexity class, and let $A \subset \mathrm{STRUC}[\sigma]$ and $B \subset \mathrm{STRUC}[\tau]$ be boolean queries. Suppose that $I\colon \mathrm{STRUC}[\sigma] \to \mathrm{STRUC}[\tau]$ is an element of $\mathcal{Q}(\mathcal{C})$ such that, for all $\mathcal{A} \in \mathrm{STRUC}[\tau]$, we have that $\mathcal{A} \in A$ if and only if $I(\mathcal{A}) \in B$. Then, I is a *$\mathcal{C}$-many-one reduction* from A to B; we say that A is *C-many-one reducible* to B, written $A \leq_{\mathcal{C}} B$.

Using this definition, we can define that a complexity class $\mathcal{C}'$ is closed under $\mathcal{C}$-reductions if $B \in \mathcal{C}'$ and $A \leq_{\mathcal{C}} B$ together imply that $\mathcal{A} \in \mathcal{C}'$. Two of the most common reductions in complexity theory are L-reductions and P-reductions,. In fact, $\mathrm{L}, \mathrm{NL}, \mathrm{P},$ and PSPACE are all closed under L-reductions, and the latter two under P-reductions. However, we focus on FO-reductions in this paper, for the extensions of FO that we consider below are all closed under FO-reductions.

4. The Limitations of First Order Queries: FO $\subsetneq$ L

We begin our descriptive analysis by showing that FO is a relatively weak class of languages; it is a strict subset of the set of logspace-computable languages (which is itself very weak, given that a logspace machine does not even have the power to copy its input onto its work tape).

Theorem 4.1. *FO $\subset$ L*

Proof. Let $\tau = \langle R_1^{a_1}, \dots, R_r^{a_r}, c_1, \dots, c_s \rangle$. Let $\varphi \in \mathcal{L}(\tau)$ and let φ determine a first-order boolean query, $I_\varphi \colon \mathrm{STRUC}[\tau] \to \{0,1\}$. By [Men15] Proposition 2.30, we can assume that φ is in prenex normal form, i.e, there is a quantifier-free α such that

$$\varphi \equiv (\exists x_1)(\forall x_2)\dots(Q_k x_k)\alpha(\overline{x}).$$

First, we demonstrate that quantifier-free FO formulae can be evaluated in logspace by induction on construction of the formulas. To do so, we go through the first four cases of Definition 2.5.

(i) If the formula is of the form $a = b$, then the machine can find where the value of a is stored and the value of b is stored, and check for equivalence bit-by-bit, since a and b are stored as binary numbers of length $\lceil \log n \rceil$.

(ii) To calculate $R_i(x_1, \cdots, x_{a_i})$, the machine moves its head to the read tape at location where the encoding of R_i in this structure begins. Recall that that R_i is encoded with n^{a_i} bits to represent every possible a_i-tuple. Thus, we can move the read head $x_1 \cdot n^0 + x_2 \cdot n^1 + \dots + x_{a_i} \cdot x^{a_i - 1}$ to the right to find the relevant position of this tuple (which is easy to compute, as all these values can be stored in $O(\lceil \log n \rceil)$ space and multiplication/addition are log-space computable), and simply return bit at this position.

(iii) If the formula is of the form $\phi \wedge \psi$, then in logspace we can clearly compute both ϕ and ψ and compute the $\wedge$ of the resultants, and can similarly compute $\neg\phi$.

Now we have essentially shown that φ is logspace computable in the $k = 0$ case. Now, suppose all queries in prenex form with $k-1$ quantifiers are computable, and we show that φ is logspace computable. So, we have that

$$\varphi = \exists x_1 \ \psi(x_1),$$

where

$$\psi(x_1) \equiv (\forall x_2)\dots(Q_k x_k)\alpha(\overline{x}).$$

Let M_0 be the machine that computes $\psi(c)$ in $O(\log n)$ space (which exists by inductive assumption). Then, we define M to be the machine that cycles through all possible values for x_1 (which it can do by going through all binary strings of length $\lceil \log n \rceil$), and run M_0 when c is replaced with each given value. If M_0 accepts any of these inputs, then let M accept. Else, let M reject. Since the extra space needed is of size $\log n$ to store the extra value, we maintain the invariant that the computation requires $O(\log n)$ space, and thus M is a log-space machine that computes φ. $\square$

The fact that this inclusion is strict is a much more difficult proof, for it involves proving a lower bound for the query PARITY, which returns a 1 if and only if there are an odd number of 1s in a given binary string. The proof that PARITY $\notin$ FO is a subtle argument that utilizes Håstad's Switching Lemma, and a fully fleshed-out version can be found in Theorem 13.1 in [Imm99]. However, it is clear that

PARITY $\in$ L, and can in fact be computed in constant space, for a machine can simply go through each index in the string and switch the value of an auxiliary bit every time it encounters a 1, returning the final result. Thus, FO $\subsetneq$ L.

5. P = FO(LFP)

Many problems that are not first-order expressible, such as PARITY and REACH, seem like they can be done by inducting on "first order steps"— in the case of PARITY, one can go index-by-index checking for 1s with "first order steps," and can go node-by-node checking for edges for REACH. This leads to the idea of expanding FO by adding the ability to induct in order to buff up logic without introducing second-order logic.

One method of achieving this gentle strengthening of first-order logic is through the LFP predicate, which allows us to take relations and "expand" them step by step until it reaches a fixed point. Let τ be a vocabulary, and let $\varphi(R)$ that be a first-order formula in $\mathcal{L}(\tau \cup \{R\})$ that can take as input a k-ary relation R and has at most k free-variables labelled $x_1, \ldots, x_k$; so, φ can be intuited as a "function" that takes k-ary relations to k-ary relations on a given structure. Now, given a structure $\mathcal{A} \in \text{STRUC}[\tau]$, we can define

$$\varphi^{\mathcal{A}}(R) = \{\langle x_1, \ldots, x_k \rangle \mid \mathcal{A} \models (\varphi(R))(x_1, \ldots, x_k)\}. \tag{5.1}$$

We call φ *monotone* if for all $\mathcal{A} \in \text{STRUC}[\tau]$ and any k-ary relations R and S on $\mathcal{A}$, we have that $R \subset S$ implies that $\varphi^{\mathcal{A}}(R) \subset \varphi^{\mathcal{A}}(S)$. Thus a monotone formula can "expand" a relation, as mentioned above, since, given any input relation, the output relation contains the input. Now, given a monotone φ and $r \in \mathbb{N}$, let $(\varphi^{\mathcal{A}})^r$ denote $\varphi^{\mathcal{A}}$ iterated r times, or, alternatively, the formula φ allowed r steps of expansion. Observe that, since there are n^k possible k-tuples for $||\mathcal{A}|| = n$ and φ is monotone, given a relation R, there exists some $r \leq n^k$ such that if $s \geq r$ then $(\varphi^{\mathcal{A}})^r(R) = (\varphi^{\mathcal{A}})^s(R)$. We call this relation $(\varphi^{\mathcal{A}})^r$ the fixed point of φ with initialization R, and define

$$\text{LFP}_{R^k x_1, \ldots, x_k} \varphi \equiv \text{ the fixed point of } \varphi \text{ with initialization } R = \varnothing \tag{5.2}$$

to be our desired "least fixed point" operator: it finds the minimal k-ary relation on $\mathcal{A}$ such that φ is fixed on that relation. In this definition, we use the subscript $R^k x_1, \ldots, x_k$ to denote the arity and name of the input relation in the formula, and the names of the free variables in φ. This definition can be proved well-defined through the following lemma:

Lemma 5.3. *Let $R \notin \tau$ be a relation symbol of arity k, and let φ be a monotone first-order formula. Then, if S is such that $\phi^{\mathcal{A}}(S) = S$, then $LFP_{R^k x_1, \ldots, x_k} \varphi \subset S$.*

Proof. We induct on r to show that $(\phi^{\mathcal{A}})^r(\varnothing) \subset S$, and thus clearly we will have the desired result. Clearly, $(\phi^{\mathcal{A}})^0 \varnothing = \varnothing \subset S$. Now, suppose the result holds for some r, and then we have, since ϕ is monotone,

$$(\phi^{\mathcal{A}})^{r+1}(\varnothing) = \phi^{\mathcal{A}}((\phi^{\mathcal{A}})^r(\varnothing)) \subset \phi^{\mathcal{A}}(S) = S$$

since S is a fixed point. Thus, $\text{LFP}_{R^k x_1, \ldots, x_k} \varphi$ is well-defined as the least set that is a fixed point of $\varphi^{\mathcal{A}}$. □

Finally, we call the formula φ *R-positive* if R occurs only positively in φ (i.e. within an even number of negation symbols, which is equivalent to no negation symbols when they are propogated through using DeMorgan's laws). Observe that an R-positive formula is monotone. Now, we can define the class FO(LFP):

Definition 5.4. Define FO(LFP) to be the languages that can be determined by first-order inductive definitions through application of the LFP operator. More concretely, FO(LFP) contains first-order logic and, when $\varphi(R)$ is an R^k-positive formula in FO(LFP), can use $\mathrm{LFP}_{R^k x_1,\ldots,x_k}\varphi$ as a new k-ary relation symbol denoting the least fixed point of φ.

This definition can be rather difficult to intuit, and thus we provide an example.

Example 5.5 (Graph Reachability)**.** Let $\tau_g = \langle E^2, s, t\rangle$ be the language of directed graphs. Now, define

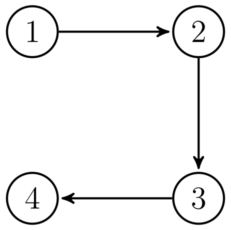
$$(\varphi(R))(x,y) \equiv (x = y) \vee \exists z(E(x,z) \wedge R(z,y)).$$

By this definition, observe that $(\varphi^{\mathcal{A}}(R))^r(x,y)$ is equivalent to "there is a path of length $\leq r$ from x to y". See the following sequence of graphs for a demonstration that the least fixed point of φ is the relation "there is a path from x to y". Thus, we shall have that

$$(\mathrm{LFP}_{R^2xy}\varphi)(s,t)$$

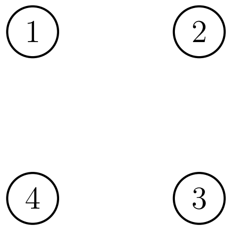
is equivalent to the reachability query exactly the reachability query (whether there is a path from start to terminal node).

Consider this for the graph G defined by

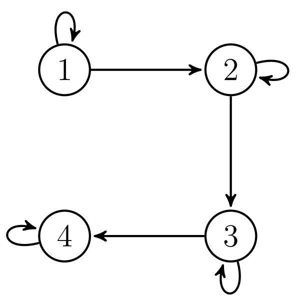


Then, we find that $(\varphi^G(\varnothing))^r$ is equivalent to, for each step r,

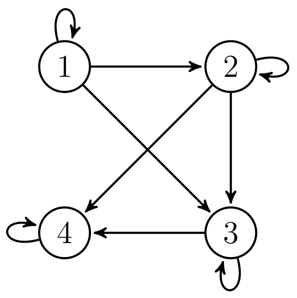
i) r = 0:



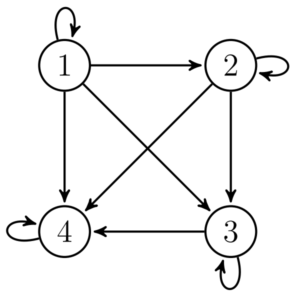
ii) r = 1:



iii) r = 2:



iv) r = 3:



Observe that $r = 3$ represents the least fixed point on this structure, and successfully produces the relation $\text{LFP}_{R^2xy}\varphi$ such that $(\text{LFP}_{R^2xy}\varphi)(x,y)$ is true if and only if y is reachable from x.

An interesting similar exercise has been left to the reader (it is not too difficult if one understands the semantics of FO(LFP))

Exercise 5.6. Let $\tau_R = \langle \cdot^3, +^3, \mathbf{1} \rangle$ be the language of commutative rings with unity. Show that, in FO(LFP), one can determine that a given $\mathcal{A} \in \text{STRUC}[\tau_R]$ satisfies the commutative ring axioms and, given elements x and y, determine whether there exists some n such that $x^n = y$ (Hint: maintain a relation R such that for some m, $R(x,y)$ is equivalent to "there exists some $k \leq m$ such that $x^m = y$") .

Now, we prove an important result regarding FO(LFP): P = FO(LFP). We do it by proving both directions in separate lemmas, for the direction P $\subset$ FO(LFP) requires a lot of work.

Lemma 5.7. *FO(LFP)* $\subset$ *P*

Proof. Consider a formula of the form $(\text{LFP}_{R^k x_1,\ldots,x_k}\varphi)$. A single first-order formula can be computed in logspace; so, we can test every k-tuple (of which there are polynomially many) in polynomial time (since L $\subset$ P). By the above, LFP reaches the fixed point after polynomially many operations, and thus in order to find the fixed point we find that it takes polynomially many total steps. This makes the full evaluation in P. Since a formula can contain only finitely many instance of an LFP operator, and is otherwise just a first-order formula, we have that FO(LFP) $\subset$ P. $\square$

For the other direction, we aim to simulate a polynomial-time computation with an FO(LFP) formula. To do so, we shall, essentially, manufacture a problem which simulates the computation of a P-machine. Note this result is well-known, but the proof constructed is original and aims to show the equality quite directly.

Lemma 5.8. *P* $\subset$ *FO(LFP)*

Proof. This proof takes inspiration from the proof that DTIME$[k^{s(n)}] \subset$ ASPACE$[s(n)]$ from [Imm99]. Let $\tau_{tm} = \langle \leq^2, \preccurlyeq^2, \text{init}^2, 0, \text{ACC}^1\rangle$. Let $M \in$ P have its time bounded by n^k, and suppose M has the alphabet Σ and set of states Q.

Suppose we have an input σ of length n to M. Now, we define a $G \in \text{STRUC}[\tau_{tm}]$ that, essentially, represents the computation of M of m. Let $K = |\Sigma \sqcup (\Sigma \times Q)|$ (where $\sqcup$ represents disjoint union), and let G have $n^k + K$ elements such that $\leq^G$ defines a total order on n^k of them (where 0^G is the minimal element) and $\preccurlyeq^G$ defines a total order on the other K. Let $\preccurlyeq^G$ be defined in some "natural" way (utilizing properties of Σ and Q, which are ordered themselves) such that this $\preccurlyeq^G$ is the same no matter the input to M; thus, these elements ordered by $\preccurlyeq$ encode $\Sigma \sqcup (\Sigma \times Q)$ in a fixed way such that ACC^G, as a monadic relation, can encode the elements representing the accepting states, which is a subset of $\Sigma \times Q$. We now want tertiary relation $C(p, t, a)$, which, if true, will mean that "tape location p has written on it a written on it at time t." This a is an element of Σ unless the tape head is on p at time t, in which case a shall be an element of $\Sigma \times Q$ and encode both what is written on the tape and the state of the head.

Let the relation $\text{init}(p, a)$ mean that the tape has a written on it at location p at initialization. Furthermore, suppose that a_{-1}, a_0, a_{+1} are such that $C(p-1, t, a_{-1})$, $C(p, t, a_0)$, and $C(p+1, t, a_{+1})$ all hold. Then, by the finite description of the Turing Machine M, we know deterministically the a such that $C(p, t+1, a)$. Let $(a_{-1}, a_0, a_{+1}) \overset{M}{\to} a$ represent this relationship between the four values (this is, in fact, a 4-ary relation that can be encoded in a first-order statement since $\Sigma \sqcup (\Sigma \times Q)$ has a fixed, finite encoding). To assist us, define "x is the predecessor of y" to be

$$\text{PRED}(x, y) \equiv x \leq y \wedge x \neq y \wedge \forall z(z \leq y \to z \leq x).$$

Similarly, we can define $\mathrm{SUCC}(x, y) \equiv y \leq x \wedge x \neq y \wedge \forall z(y \leq z \rightarrow x \leq z)$ to be the analogous successor relation. Then, we can define

(5.9)

$$(\varphi(C))(p, t, a) \equiv (t = 0 \wedge \mathrm{init}(p, a)) \vee \exists t_{-1} \Bigg(\mathrm{PRED}(t_{-1}, t) \wedge$$
$$\exists p_{-1}, p_{+1}, a_{-1}, a_{+1}, a_0 \Big(\mathrm{PRED}(p_{-1}, p) \wedge \mathrm{SUCC}(p_{+1}, p) \wedge$$
$$C(p_{-1}, t_{-1}, a_{-1}) \wedge C(p, t_{-1}, a_0) \wedge C(p_{+1}, t_{-1}, a_{+1}) \wedge \Big((a_{-1}, a_0, a_{+1}) \overset{M}{\rightarrow} a \Big) \Big) \Bigg).$$

note this equation only handles positions p that are not at the beginning or end of the tape; these edge cases can be handled in a similar fashion and added to this expression for completeness. Under close inspection, one finds that $\varphi(C)$, essentially, progresses the calculation by a single step. To see this, suppose that for some r, we have that $(\varphi^G)^r(\varnothing)$ can "compute" the first r steps of the calculation (so, $C(p, t, a)$ is correct for all $t \leq r$). Then, we see that this predicate essentially asks: given the t_{-1}, the time step before t, its neighboring cells p_{-1} and p_{+1}, and the states of p_{-1}, p, p_{+1} respectfully represented as a_{-1}, a_0, a_{+1}, do we find that p, at time t, should be encoded with a? Given this, we find that $\mathrm{LFP}_{C^3pta}\varphi$ encodes the computation. Therefore, once this is complete, we only have to ask whether the computation ever reaches an accepting state. Thus, we can write the FO(LFP) query

$$I_\sigma \equiv \exists p' t' a' \left(\mathrm{ACC}(a') \wedge (\mathrm{LFP}_{C^3pta})(p', t', a') \right),$$

knowing that M accepts input σ if and only if $G \vDash I_\sigma$.

Observe that this does not demonstrate, yet, that P $\subset$ FO(LFP). But, the construction of the graph above can be done using a k-ary first-order reduction from the given binary string input into the appropriate graph; thus, we have essentially constructed a problem in FO(LFP) to which every polynomial-time computable query is first-order reducible, and so, since FO(LFP) closed under first-order reductions (which is a fairly intuitive result), we find that P $\subset$ FO(LFP).

□

Thus, we have the following result:

Theorem 5.10. *P = FO(LFP).*

Intuitively, this makes a fair amount of sense, for FO(LFP) is able to expand FO with an operation that is fairly powerful but, simultaneously, has a "polynomial" upper bound since it necessarily terminates after n^k steps. This intuitive correlation becomes even more clear given the next section, in the way that PFP lacks this natural polynomial upper bound and can, hypothetically, try every possible configuration of the space.

6. PSPACE = FO(PFP)

Consider the types of formulas discussed in the previous section: they, essentially, took k-ary formulas and returned k-ary formulas. Above, we required that these formulae were monotone in order to ensure that they reached a fixed point in polynomially many steps; however, what if we removed this restriction? In that case, we have no assurance that the formulae will ever hit a fixed point, and might infinitely cycle. Thus, we define, for any such φ in a structure $\mathcal{A}$,

$$(\mathrm{PFP}_{R^k x_1,\dots,x_k}\varphi)^{\mathcal{A}} = \begin{cases} (\varphi^{\mathcal{A}})^r(\varnothing) & \text{there exists some } r \text{ such that } (\varphi^{\mathcal{A}})^r(\varnothing) = (\varphi^{\mathcal{A}})^{r+1}(\varnothing) \\ \varnothing & \text{there exists no such } r. \end{cases}$$

Thus, we see that the definition of PFP allows us to expand the definition of LFP without worrying about getting snagged in an infinite loop. We define FO(PFP) similarly to FO(LFP).

Definition 6.1. Define *FO*(*PFP*) to be the the analogue of FO(LFP), except, instead of allowing arbitrary instances of the LFP operator, we allow arbitrary instances of the PFP operator and non-monotone inputs φ.

The main distinction of PFP from LFP is that PFP allows us to permute the space and not "pay" for our previous steps since the formula need not necessarily be monotone— much like the distinction between PSPACE and P. We now demonstrate their equivalence formally.

Lemma 6.2. *FO*(*PFP*) $\subset$ *PSPACE.*

Proof. As above, this is the easier direction of the equivalence. Suppose we have a structure $\tau = \langle R_1^{a_1}, \dots, R_r^{a_r}, c_1, \dots, c_s\rangle$ and $\mathcal{A} \in \mathrm{STRUC}[\tau]$, and, for some formula φ, would like to evaluate $(\mathrm{PFP}_{R^k x_1,\dots x_k}\varphi)^{\mathcal{A}}$. Then, we will let the first $2 \cdot m^k$ (where $m = ||\mathcal{A}||$) elements of the work tape represent the status of the relation φ and the previous status. So, in the context of evaluating $\varphi(R)(x_1, \dots, x_k)$, the first m^k elements of the work tape are used to evaluate this relation, and the second m^k will hold the information that denotes R.

From there, we can determine the value of $\varphi(R)(x_1, \dots, x_k)$ using the stored value for R and an extra, logarithmic amount of space to evaluate the first-order expression φ. After this has been done for every tuple $x_1, \dots, x_k$, we check if we are at a fixed point, or if we have run the simulation $2^{m^{k+1}}$ times; in the former case, we use the relation just calculated, and in the later return the empty relation. Otherwise, move the first m^k elements on the tape over by m^k locations, and run again. There are fewer than m^{k+1} k-tuples, and thus $2^{m^{k+1}}$ possible k-ary relations; so, if we run for more than $2^{m^{k+1}}$ iterations, we are necessarily in an infinite loop. This can easily be tracked in an m^{k+1}-bit number.

From here, note that, since FO(PFP) consists of first order statements peppered with the PFP operator, we only have to do this polynomial-space operation finitely many times, and thus the entire operation can be done in polynomial space. $\square$

For the other direction of the proof, we proceed similarly as the proof of Lemma 5.8 above.

Lemma 6.3. *PSPACE* $\subset$ *FO*(*PFP*).

Proof. Let M be a PSPACE machine, and suppose that, once it reaches q_{accept} or q_{reject}, has an "identity" transition function (so, nothing happens each step). If the machine does not terminate, consider it to reject the input. We ensure the machine either accepts and reaches a fixed point, rejects and reaches a fixed point, or cycles indefinitely.

From here, we can construct the "computation" graph $G \in \tau_{tm}$ to be equivalent to the one described in Lemma 5.8. Just as above, this is a first-order reduction and thus, as FO(PFP) is closed under first-order reductions, demonstrating that this reduction is valid completes the proof. However, this time, we do not care about how many times steps we take; so, we shall only keep track of a tuple $C(p, a)$ which demonstrates that "a is written on tape position p." To assist us, for a k-ary relation R^k, let $\text{EMPT}_k(R) \equiv \forall x_1, \ldots, x_k \quad \neg R(x_1, \ldots, x_k)$. This is equivalent to asking if $R = \varnothing$. Thus, we can write our recurrence

$$\begin{aligned}(\varphi(C))(p,a) \equiv (\text{EMPT}_2(C) \wedge \text{init}(p,a)) \vee \exists p_{-1}, p_{+1}, a_{-1}, a_0, a_{+1}\Big(\\ \text{PRED}(p_{-1},p) \wedge \text{SUCC}(p_{+1},p) \wedge C(p_{-1},a_{-1}) \wedge C(p,a_0) \wedge \\ C(p_{+1},a_{+1}) \wedge ((a_{-1},a_0,a_{+1}) \overset{M}{\to} a)\Big)\end{aligned}$$

Thus, $\varphi(C)$ advances the computation by a single step, without caring about the time-step. Observe that it never returns the empty relation, given well-formed input. Thus, to figure out whether the computation accepts the input, we need only ask

$$I_\varphi \equiv \exists p' a' \left(\text{ACC}(a') \wedge (\text{PFP}_{C^2 pa}\varphi)^G(p,a)\right).$$

We reject if the machine loops infinitely, and, since we assumed any fixed point of the computation is either an acceptance or rejection state, I_φ defined on input σ is equivalent to M's behavior on input σ. Therefore, PSPACE $\subset$ FO(PFP). □

Thus, we get to the main result of this section,

Theorem 6.4. *PSPACE = FO(PFP).*

7. Transitive Closure: NL = FO(TC) and L = FO(DTC)

In the previous two sections, we utilized LFP and PFP's abilities to take a relation and iterate on it— that is, through first order steps, inductively use a relation to define a new relation. However, in the quest to characterize NL and L similarly, we run into a problem: we do not have enough space to store a new relation, for it takes polynomial amount of space to encode an arbitrary relation. However, we are able to compute first-order formulas in logarithmic space; so, given logspace, we are able to compute a fixed first order formula and iterate through possible input arguments. In order to assist in this intuition, let $\tau_g = \langle E^2 \rangle$ be the vocabulary of graphs with no labelled nodes, and, given a formula $\varphi(\overline{x}, \overline{x}')$ that takes in as input a pair of k-tuples, we define a k-ary reduction $G_\varphi \colon \text{STRUC}[\tau] \to \text{STRUC}[\tau_g]$, where, for a structure $\mathcal{A} \in \text{STRUC}[\tau]$, we define $G_\varphi(\mathcal{A})$ to be a structure $G \in \text{STRUC}[\tau_g]$ such that

$$V^G = |\mathcal{A}|^k, \qquad E^G = \{\langle \overline{x}, \overline{x}' \rangle \mid \varphi(\overline{x}, \overline{x}')\}.$$

Then, given a machine with logarithmic space, we can, essentially, walk node-to-node in $G_\varphi(\mathcal{A})$ and determine which node are neighbors, for it takes logarithmic space to store $\langle \overline{x}, \overline{x}' \rangle$ and compute whether $\varphi(\overline{x}, \overline{x}')$. This naturally brings us upon our next definition, which relates this ability to go "node by node" along a first-order relation.

Definition 7.1. For a first-order formula $\varphi \in \mathcal{L}(\tau)$ of arity $2k$ and $\mathcal{A} \in \text{STRUC}[\tau]$, we define $(\text{TC}_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}$ to be the reflexive, transitive closure of φ; i.e., if $\phi \equiv (\text{TC}_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}$, then ϕ is the minimal relation such that

$$\forall \overline{x}, \overline{x}' \quad (\varphi(\overline{x}, \overline{x}') \rightarrow \phi(\overline{x}, \overline{x}')) \wedge \forall \overline{x} \quad \phi(\overline{x}) \wedge$$
$$\forall \overline{x}, \overline{x_0}, \overline{x}' \quad (\phi(\overline{x}, \overline{x_0}) \wedge \phi(\overline{x_0}, \overline{x}') \rightarrow \phi(\overline{x}, \overline{x}'))\,.$$

We prove a lemma about TC that provides some intuition about it, allowing us to connect it to NL.

Lemma 7.2. *$(TC_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}(\overline{y}, \overline{y}')$ holds if and only if there exists a path from $\overline{y}$ to $\overline{y}'$ in $G_\varphi(\mathcal{A})$.*

Proof. For simplicity, let $\phi = (\text{TC}_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}$.

Suppose there exists a path from $\overline{y}$ to $\overline{y}'$ in $G_\varphi(\mathcal{A})$. Then, there exist $\overline{y} = y_0, y_1, \dots, y_k = \overline{y}'$ such that $\varphi(y_i, y_{i+1})$ for all i. Then, necessarily, $\phi(y_0, y_1)$ holds and $\phi(y_1, y_2)$ holds, and thus, $\phi(y_0, y_2)$ holds. By induction, one can show that $\phi(y_0, y_i) = \phi(\overline{y}, \overline{y}')$ holds.

Let $\psi(y, y')$ be the relation "there exists a path from y to y' in $G_\varphi(\mathcal{A})$." Then, clearly we have $\forall x \ \ \psi(x)$, since all nodes are reachable from themselves. Further, the path relation is transitive, and thus $\forall x_0, x_1, x_2 \ \ (\psi(x_0, x_1) \wedge \psi(x_1, x_2) \rightarrow \psi(x_0, x_2))$. Finally, if $\varphi(x, y)$ holds, then there is an edge from x to y in $G_\varphi(\mathcal{A})$, and so there is trivially a path from x to y, and thus $\psi(x, y)$ holds. By definition, $\phi \subset \psi$, and therefore if $\phi(x, y)$ holds then there exists a path from x to y in $G_\varphi(\mathcal{A})$. □

So, if we want to determine whether $(\text{TC}_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}(\overline{x}, \overline{y})$ holds, we can picture a first order reduction to $G_\varphi(\mathcal{A})$ and then ask whether there exists a path from $\overline{x}$ to $\overline{y}$ (often phrased $\overline{y}$ is *reachable* from $\overline{x}$). This is a natural problem to solve in NL: one can store the description of a node in logarithmic space, then nondeterministically check every other node to see if it is a neighbor and, if so, recurse by doing the same for that node. By having initial input x and constantly checking if the input becomes y, we can see whether there exists a path.

Similarly to above, we define FO(TC) to be the extension of first-order logic by arbitrary occurrences of the TC operator. However, this poses a problem; in the described method, NL can naturally capture an instance of $(\text{TC}_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}(x, y)$, but cannot capture $\neg(\text{TC}_{x_1,\dots,x_k,x'_1,\dots,x'_k}\varphi)^{\mathcal{A}}(x, y)$, since the construction of NL naturally represents a $\exists$ form of predicate, not a $\forall$. Thus, we let FO(pos-TC) represent the subset of of FO(TC) where all the instance of TC occur within a positive number of negation symbols, and thus we can easily recursively use NL to compute all these instances in the above method just mentioned. Thus, it is natural to find

Lemma 7.3. *FO(pos-TC) $\subset$ NL*

Proof. Observe that all relations computable in NL are closed under $\forall$ and $\exists$ quantifiers, for we can deterministically cycle through all possible values in $\log n$ space. Thus, we need only show that, if $\varphi(\overline{x}, \overline{x}')$ is computable in NL, then $(\text{TC}_{\overline{x},\overline{x}'}\varphi)$ is also computable in NL. Suppose we would like to compute $(\mathrm{TC}_{\overline{x},\overline{x}'}\varphi)(\overline{a},\overline{a}')$. If $\overline{a}=\overline{a}'$, then accept. Otherwise, guess some $\overline{y}$, check if $\varphi(\overline{a},\overline{y})$ holds and, if so, recurse and guess some $\overline{z}$, check if $\varphi(\overline{y},\overline{z})$ holds, and continue this process unless eventually, for an input $\overline{w}$, we guess $\overline{a}'$ and find that $\varphi(\overline{w},\overline{a}')$ holds; if so, return true. This procedure, it can be seen, can compute the predicate "does there exist a path from $\overline{a}$ to $\overline{a}'$ in $G_\varphi(\mathcal{A})$," and thus properly computes $(\mathrm{TC}_{\overline{x},\overline{x}'}\varphi)(\overline{a},\overline{a}')$. By induction, we find that all pos-TC formulas are in NL. □

For the other direction, we proceed quite differently than the proofs that P $\subset$ FO(LFP) and PSPACE $\subset$ FO(PFP) and utilize the fact that there are polynomially many states for a log-space machine.

Lemma 7.4. *NL $\subset$ FO(pos-TC)*

Proof. Below, we construct a first order reduction from the binary string input of the machine to an FO(pos-TC) query such that the FO(pos-TC) query holds if and only if the machine accepts the input. Since FO(pos-TC) is closed under first-order reductions, this will imply NL $\subset$ FO(pos-TC).

Suppose we have an NL machine M, whose work tape is space-bounded by $k\log n$ (where n is the size of the input). Without loss of generality, we can assume that the alphabet of M consists of only $\{0,1\}$. This implies there are $2^{k\log n}=n^k$ possible states for the work tape. Now, suppose that σ is an input to M. Through a first-order reduction from the language of binary strings, we can can construct a graph $G\in\mathrm{STRUC}[\tau]$ of size n^{k+1} for $\tau=\langle\mathrm{init}^1,\leq^2,\mathrm{BIT}^2,\rangle$ such that $\mathrm{init}(x)$ holds if the xth element of the input tape is a 1, and a 0 otherwise. Now, since the work tape has size $k\log n$ and is written with 0s and 1s, and each value in τ can be coded as a $\log(n^{k+1})=(k+1)\log n$ bit number, a single element can be read as storing both the work tape and the location of the head on the read tape (and we can access the read tape using init). Further, we can add a constant number of bits at the beginning of the value to represent the state of the read head.

As the transition function of M is finitely describable, it is possible to write a first-order formula $\varphi(x,y)$ that holds if and only if y represents a possible next step in the computation after x. Thus, $(\mathrm{TC}_{x,y}\varphi)(x',y')$ holds if and only if there is a computation path that takes x' to y'. We can construct a first-order formula $\mathrm{ACC}(x)$ with no negation symbols such that $\mathrm{ACC}(x)$ holds if and only if x is an accepting state (since the accepting state is coded as a finite number of bits at the beginning of the elements), and therefore, since the read head is initially at location 0 and the read tape is initialized to 0 (and thus the initialization can be represented by the 0 bit string), we find that the computation accepts the initial input if

$$\exists x'y'\,(\forall k\neg\mathrm{BIT}(x',k)\wedge\mathrm{ACC}(y')\wedge(\mathrm{TC}_{x,y}\varphi)(x',y'))$$

holds. This is an FO(pos-TC) formula since the TC occurrence is not within any negation symbols. Thus, we find that this query accepts the input if and only if the original NL machine would. □

Through a rather detailed proof, Theorem 9.20 in [Imm99] demonstrates that for finite, ordered structures, FO(pos-TC) = FO(TC); thus, we can expand our results to find

Theorem 7.5. *NL = FO(pos-TC) = FO(TC).*

Comparing with our characterizations for P and PSPACE, note that, through a first-order reduction, one can only make the size of the image structure polynomial in the size of the domain structure. Therefore, for the NL $\subset$ FO(pos-TC) proof, we were able to code each possible state of the work tape as a single element of the new structure. Because of that, we only had to use the "is a possible next state" relation and find a path to an accepting state in order to determine whether the NL machine would accept. This is more difficult with P; for a tape of length n^k there are 2^{n^k} possible states, and thus we cannot store all states in the image structure of a first-order reduction with such ease, and need to store the state as a relation amongst the elements, and therefore, need to recompute the "is a next step of" relation step by step through the LFP operator. This is why the method of Lemma 7.4 would not work to demonstrate P $\subset$ FO(TC) (though, at this writing, there is no proof this relation is not true).

Further, the method of Lemma 7.3 cannot be used to show that FO(pos-TC) $\subset$ L; we utilized that finding existence of a path is quite natural for a machine that can "guess" a next node. But, what if each node in a graph had out-degree at most one? Then, a log-space machine could determine existence of a path similarly to Lemma 7.3, for, knowing that the next node in the path must be uniquely determined, it could simply continually choose the unique next node in the path, and wait to hit the target last node. In this description, one must keep track of the initial node and, if this node is arrived at again, return that there does not exist a path (for one must be wary of cycles in such a deterministic computation). Therefore, we define the *deterministic reduct* of a formula φ to be

$$\varphi_d(\overline{x}, \overline{y}) \equiv \varphi(\overline{x}, \overline{y}) \wedge [(\forall \overline{z}) \neg \varphi(\overline{x}, \overline{z}) \vee (\overline{y} = \overline{z})].$$

By this definition, φ_d is the subrelation of φ such that $\varphi_d(\overline{x}, \overline{y})$ holds if and only if $\varphi(\overline{x}, \overline{y})$ holds and, for all $\overline{z}$ not equal to $\overline{y}$, $\varphi(\overline{x}, \overline{z})$ does not hold. In this sense, φ_d forces φ down to a "deterministic" relation; thus, it is natural to define the *deterministic transitive closure* of φ, or (DTCφ), such that

$$(\mathrm{DTC}\varphi) \equiv (\mathrm{TC}\varphi_d).$$

From here, we define FO(DTC) similarly to FO(TC) above. Using, essentially, the same proofs as in Lemmas 7.3 and 7.4, we leave it to the reader to utilize this determinism and demonstrate that

Theorem 7.6. *L = FO(DTC)*

Unlike above, we do not need to work with pos-DTC since L, as a deterministic class, is closed under complementation.

8. Further Thoughts

As we have seen, the concept of adding "induction," in various forms, to first-order logic gives it the ability to capture the complexity classes P, PSPACE, NL, and L. However, this brings about the question: are there other forms of induction that allow us to capture other Turing Machine-based complexity classes, such as NP, co-NP, or the polynomial hierarchy? Some classes already have descriptive definitions (the result that NP $=$ SO$\exists$ was the one that began the field of descriptive complextiy!), but it would be nice to see a characterization in the mode discussed in this paper. Is it possible to even capture classes such as AC^0, NC^1, or other circuit based classes? If so, it could give us key insight on the relationships between many different classes, many of which have very distinct definitions. Furthermore, could this route be used to provide lower bounds? If one were to demonstrate that a problem solvable in P cannot be determined in FO(DTC), then they would have proven that L $\subsetneq$ P; if one finds a characterization of NP and shows it is equivalent to or distinct from FO(LFP), they will have resolved P $\overset{?}{=}$ NP.

Acknowledgments

I would like to thank Neil Immerman for providing useful input in determining the topic of this paper, Alexander Razborov for helping me choose a subject for this paper and for teaching me all the complexity theory I needed to pursue it, and Peter May for putting together the REU program and providing edits on this paper. A special thanks goes to Colin Aitken, my REU mentor, for helping me as I stumbled my way through many a book and paper and for often joining me in cluelessly staring at a chalkboard. Furthermore, I would like to thank my family and friends for listening to me go on and on about about my pet project, and the librarians at the University of Chicago libraries for letting me check out a (semi)-ridiculous number of books at a time as I learned enough logic to understand this topic.

References

[1] [Aro09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* New York: Cambridge University Press, 2009.

[2] [Imm99] Neil Immerman. *Descriptive Complexity.* New York: Springer-Verlag, 1999.

[3] [Men15] Elliott Mendelson. *Introduction to Mathematical Logic.* 6th ed. Boca Raton, Florida: CRC Press, 2015.