The Expressiveness of a Family of Finite Set Languages

Neil Immerman   Sushant Patnaik
David Stemple

COINS Technical Report 91-33 *

* A shortened version of this paper appears in the Proceedings of the 1991 ACM Symposium on Principles of Database Systems

The Expressiveness of a Family of Finite Set Languages

*Neil Immerman** *Sushant Patnaik** *David Stemple*†

Computer and Information Science Department
University of Massachusetts
Amherst, MA 01003

Abstract

In this paper we characterise exactly the complexity of a set based database language called *SRL*, which presents a unified framework for queries and updates. By imposing simple syntactic restrictions on it, we are able to express exactly the classes, *P* and *LOGSPACE.* We also discuss the role of ordering in database query languages and show that the *hom* operator of *Machiavelli* language in [OBB89] does not capture all the order-independent properties.

1. Introduction

The expressiveness and complexity of database query and transaction languages are of interest for a number of reasons. Since the size of inputs to expressions in these languages is often very large, controlling the expressiveness of a language can be used to reduce the number of intractable queries posed by naive users, a major clientele of query languages. In addition, powerful optimization techniques are easier to develop and apply to limited languages than to more general languages. It is also often easier to reason formally about limited languages than about more general languages, though it is sometimes hard to isolate the difficulties stemming from the superficial

*Research supported by NSF grant CCR-9008416

†Research supported by NSF grants IRI-8606424 and IRI-8822121 and ONR U.R.I. grant N00014-86-K-0764.

diversity of a language, i. e., a large number of ways of expressing the same computation, from those due to its computational complexity. Our motivation includes the first two reasons, but is also strongly concerned with the third - the tractability of reasoning about finite set computations. While such tractability can be useful in optimizing queries and transactions, it can also be used to assure the quality of systems, for example, in terms of consistency maintenance over transactions.

Here we address the expressiveness of languages for specifying computations over finite sets. The family of languages we consider has very few primitives and its semantics are expressed by a small set of algebraic axioms. It does not start with first order logic and set theory, nor rely on concepts of destructive update or random access memory (or the related concepts of object ids and *ref* types). Unlike many approaches to computing with finite sets, it is designed to be seamlessly combined with other algebraic computational models such as ordinary arithmetic or recursive data type algebras. One of our goals is to be able to reason effectively about the complexity *and other properties* of computations over combined algebras, including finite set algebra.

Our logic base is simply the *if-then-else* function. First order logic is included in our computational model as a result of combining set traversal and the *if-then-else* operator. Our framework includes simple *tuple algebra*, expressing the ability to *construct* typed, fixed arity, non-recursive tuples and to *select* their components. Set traversal is expressed using a single mechanism, the higher order function *set-reduce*, which applies functions as it traverses a set. This is the sole iterative construct, and it can depend on the order of traversal. Its formalization in algebraic axioms makes the order of elements in a set manifest and allows order dependence to be reasoned about using our mechanical reasoning capabilities, which are based on Boyer-Moore computational logic [BM]. In this way, we can often prove that the result of a computation is order independent even though the ordering is implicitly used as we traverse a set. This allows a new approach to a question raised by Chandra and Harel as to whether there is a language that expresses exactly the polynomial-time, order independent queries. All previous research on polynomial time queries has chosen either to deal with languages that express order dependent queries, or for which certain simple order independent queries cannot be expressed.

There have been numerous studies of the expressive power of query languages. For instance, it is well known that first order relational query languages are limited in their expressibility [AU79]. However when augmented with recursion or looping (as an added primitive) they become sufficiently powerful to express exactly the queries in various complexity classes. Characterizing the expressive power of such languages has been the principal object of study in [Va82], [CH80], [CH82a], [CH82b], [AU79], [Imm82], [AV89]. For example, Immerman and Vardi discovered independently that fixpoint logic plus ordering expresses the set of polynomial time computable queries [Va82], [Imm82].

A common and rather useful way of expressiveness is to use complexity characterizations. One finds classes of queries capturing $LOGSPACE, P$, $PSPACE$, $PRIMREC$. Interestingly, most of the classes of queries considered turned out to capture some complexity class. It seems that certain query language comparisons are connected with deep problems of complexity theory. Recently parallel evaluation of recursive queries has also drawn considerable attention in [CK85], [AC89].

In the past the emphasis has been to develop a *natural* set of primitives for a query language so that it can compute all the *computable* queries as in [CH80], [Ch81], [AV88]. Unbounded arity relations or the ability to create new values have been used. For example, Chandra and Harel, in [CH80], define the concept of *computable* queries and present a *complete* database programming language and show that relational algebra augmented with the power of iteration is *complete.* [1] In [HS89b], Hull and Su consider a hierarchy of languages whose complexity is in the super exponential range. However, we are interested in devising a *natural* language whose complexity is clear from the syntax but for *feasible* complexity classes from a database point of view, e.g. $TIME[n]$ and $SPACE[log^k n]$. Instead of *computable queries*, we regard primitive recursive queries as the high end of the spectrum. Indeed, all of the interesting complexity classes are contained in *PrimRec.* Our measure of complexity is data-complexity as defined by Vardi [Va82].

The *set-reduce* construct (defined in section 2) can be thought of as a bounded loop primitive. See [SS89] for more details. The *set-reduce* construct resembles the *hom* operator of the database programming language called *Machiavelli* [OBB89]. We define the transaction language, *unrestricted SRL* and show that its corresponding query language captures the primitive recursive properties. We then show that natural restrictions of this language capture P, $DSPACE(\log n)$ and $NSPACE(\log n)$.

The expressive power of the bounded loop construct or its variant has been studied before in [AU79], [Va82], [Q89], [CH80], [Imm87], [AV88] but not in this framework. In [Ch81], Chandra raises the question of specifying a set of primitives of the form *forall tuples t in relation R do statement S*, where S is restricted so as not to use the order in which the *forall* cycles over all the tuples, such that programs in this style can compute all the computable queries. The *set-reduce* construct provides a partial answer.

In [AV88], Abiteboul and Vianu present declarative and procedural update languages and show that they are *complete* in Chandra and Harel's sense. They also

[1] Here a complete database language is one that can compute every partial recursive function of its database [CH80].

define restrictions on their languages and characterize their expressiveness. They define the so-called "*non-deterministic*" updates and show certain languages to be "*non-deterministic*" update complete. Their definition of "*non-deterministic*" updates is actually what we refer to as *order-dependent*. It should not be confused as such with *non-determinism* as referred to in complexity theory. In [Q89], Qian studies the complexity of a bounded looping construct *foreach x in R/p(x) do t(x)* and shows that, under deterministic semantics, her language and a subclass of it have polynomial time and first order expressive power. Her looping construct closely resembles the *set-reduce* operator, however the two corresponding languages differ in their algebras. Her definition of "*non-deterministic*" semantics, identical to Abiteboul and Vianu's, does not lead to a decidable distinction between non-deterministic and deterministic languages. In a recent paper [AK90], Abiteboul and Kanellakis discuss an object oriented database programming language wherein objects are built by applying *set* and *tuple* constructors. They define an algebra for their language which is built from first-order operators augmented with the *powerset* operator. The latter immediately puts the data-complexity of their language in the exponential range. Had they instead defined a bounded iterator operator, as we do, then it would have been possible to derive sub-languages whose complexities lie between first-order and exponential.

This paper is organized as follows. Section 2 defines the *set-reduce* language and gives some background. Section 3 presents some tools of descriptive complexity [Imm87] and proves that *SRL* (with set-height at most 1) = P. Section 4 describes ways of restricting the complexity of *SRL*. Section 5 shows that *unrestricted SRL* with sets of unbounded width (or, equivalently, *SRL* with invented values) captures the class of primitive recursive functions. Section 6 shows how to deduce the complexity of a *SRL* program from its syntax. Section 7 discusses the role of ordering in database query languages. Section 8 concludes with some comments and open questions.

2. The language

SRL is a typed language of finite expressions constructed and typed according to the following rules:

1. *true* and *false* of type *boolean* are SRL expressions.

2. *if srebool then* sre_1 *else* sre_2, where *srebool* is an SRL expression of type *boolean*, and sre_1 and sre_2 are SRL expressions of the same type, is an SRL expression of the same type as sre_1 and sre_2.

3. constants of type T where T includes an equality relation are SRL expressions of type T.

4. $[sre_1, ..., sre_n]$ of type $tuple(sel_1 : T_1, ..., sel_n : T_n)$, where T_i is the type of sre_i, is an SRL expression of type $tuple(sel_1 : T_1, ..., sel_n : T_n)$.

5. $sel_i(sre)$ where sre is of type $tuple(sel_1 : T_1, ..., sel_n : T_n)$ is an SRL expression of type T_i.

6. $sre_1 = sre_2$, where sre_1 and sre_2 SRL expressions of the same type, is an SRL expression of type *boolean*.

7. *emptyset* is an SRL expression of type $set(alpha)$ where alpha matches any type.

8. $insert(e, s)$ for s an SRL expression of type $set(T)$ and e of type T is an SRL expression of type $set(T)$.

9. *set-reduce* $(s, app, acc, base, extra)$ is an SRL expression of type T', where s, *base* and *extra* are SRL expressions of types $set(T)$, T' and *extype*, respectively, and *app* and *acc* are formed by appending $lambda(x, y)$ to SRL expressions, in which only x and y can appear free. The variables x and y in *app* must appear in places appropriate for SRL expressions of types T and *extype*; and in *acc* in places appropriate for T and T', respectively. The typing of lambda expressions follows the normal type inference rules for lambda expression applications.

10. $(srlexp)$ is an SRL expression if *srlexp* is an SRL expression, and it has the same type as *srlexp*.

The semantics of SRL is given by the following rules and equations for which there is a straightforward reduction mechanism that is complete for deciding equality of ground terms.

Boolean

$not(true = false)$
$(if\ true\ then\ e_1\ else\ e_2) = e_1$
$(if\ false\ then\ e_1\ else\ e_2) = e_2$

Other types

The equality of constants of types other than boolean, tuple and set is defined by the types.

Tuples

Tuple construction is a function.
For tuples t and t' of type $tuple(sel_1 : T_1, ..., sel_n : T_n)$,
e_i typed T_i for $i = 1$ *to* n, $t = [e_1, ..., e_i, ...e_n]$
and $t' = [e'_1, ..., e'_i, ...e'_n]$

$(t = t') -> (e_i = e'_i)$ for $i = 1$ *to* n

$sel_i(t) = e_i$ for $i = 1$ *to* n

Finite sets
The semantics of *emptyset*, *insert*, *choose* and *rest* (the latter two used in the semantics of *set-reduce*) are given in [SS89].

```
set-reduce(s,app,acc,base,extra) =
 if s = emptyset
    then base
    else acc(app(choose(s),extra),set-reduce(rest(s),app,acc,base,extra))
```

The semantics of lambda expressions are given by straightforward reduction rules.

Parentheses
$(srlexp) = srlexp$

The above specifies a many-sorted signature and a set of axioms. The axioms for finite sets specify the existence of a total order on the domain type of a finite set type. Any model (algebra) of the specification must supply an order. If the type of the set elements has any structure (relations or functions) other than equality, such as tuple structure, certain expressions in the language will differ from one model to another because of this. (Finite sets of types with only equality have an initial algebra, but adding any function of the element type other than equality to the signature destroys the initiality of the specification.)

The class of *set-reduce functions* is the smallest class of functions computed by such *SRL* expressions and closed under *composition* and *set-reduce* operations. Denote it as *unrestricted* SR.

Note that boolean *and, or*, and *not* can easily be defined with the *if-then-else* function. Also, note that we have made available to us an ordering relation (denoted by $\leq$) on the domain which is the same order in which the elements are scanned by the *choose* mechanism of *set-reduce*. This is quite natural, since any computation must use an ordering. See Section 7 for a discussion of the ordering.

We believe that the *SRL* framework can provide a suitable base on which algebraic specifications of computations over databases can be analysed for purposes of assuring correct behavior and achieving optimized implementations. We wish to limit the expressiveness of *unrestricted SRL* to within *reasonable* complexity classes so as to make the latter task feasible. We impose syntactic restrictions on *unrestricted SRL* and study their effect on its expressive power. Define *set-height*() as follows:

set-height(*base-type*) = 0
set-height(*set of* α) = 1+*set-height*(α)

As a first step, we restrict the use of *set types* to those with *set-height* ≤ 1, but allow arbitrary, though fixed, nesting and width of *tuple types*. Let us denote this restricted version of the language as *SRL* and define the class of decision problems expressible in *SRL* as *SR*. Functions in *SRL* are similar to Cobham's recursive functions [Co64] and we show that the two classes are indeed equivalent. The restrictions of ***set-height*** and ***tuple-width*** are, as shown later, quite crucial to our result. To get started use the following fact:

Fact 2.1 ([**SS89**]) *Finite set functions such as union, intersection, difference, membership; predicates for universal and existential quantification such as forall, forsome; and relational operators such as join, project and select can be expressed in SRL.*

3. Expressiveness of SRL

Definitions.

Our approach to characterizing the expressive complexity of *SRL* follows the conventions of descriptive complexity [Imm87]. We will code all inputs as finite logical structures. The universe of structure A is $\{0, 1, \ldots, n-1\}$ and is denoted by $|A|$. A vocabulary $\tau = < R_1^{a_1}, R_2^{a_2}, \ldots, R_s^{a_s} >$ is a tuple of input relation symbols of fixed arities. Let $STRUCT[\tau]$ denote the set of all finite structures of vocabulary τ. We will think of all complexity theoretic problems as subsets of $STRUCT[\tau]$ for some τ. The advantage of this approach is that when we consider our inputs as first order structures we may write properties of them in variants of first-order logic.

For any vocabulary τ, there is a corresponding first-order language $L(\tau)$ built up from the relation symbols of τ and the logical relation symbols and constant symbols

: $=, \leq, 0, n-1$, using logical connectives : $\vee, \wedge, \neg$, variables : $x, y, z, ..$, and quantifiers : $\forall, \exists$. Let FO be the set of first-order definable problems:

$$FO = \{S | (\exists \tau)(\exists \varphi \in L(\tau)) S \in STRUCT[\tau] \models \varphi\}.$$

Let us recall the definition of *first-order interpretation* [IL89, Imm87]. Let $S \subset STRUCT[\sigma]$, $T \subset STRUCT[\tau]$ be two problems. For simplicity, assume that the vocabularies, τ, σ consist of single input relations, $< R^b >, < Q^a >$ of arity b and a, respectively.

For example, consider the following problem, which will prove to be useful later. Let S_n denote the group of permutations on $1, 2, \ldots, n$ under composition. Let IM_{S_n} denote the following iterated multiplication problem: given permutations $s_1, \ldots, s_n \in S_n$ as input, compute their composition, i.e. $s_1 * s_2 * \cdots * s_n$. The input structure can be encoded as n^3 bits by a 3-ary relation: $R(i, j, k)$ which equals 1 iff the i^{th} permutation sends j to k.

Let $k \geq 1$ be a constant and let $\varphi(x_1, \ldots, x_k)$ be a FO formula from $L(\sigma)$. Then φ defines a mapping m_φ from $STRUCT[\sigma]$ to $STRUCT[\tau]$. Let $A = < n, Q^a > \in STRUCT[\sigma]$ be a string of length n^a. Then $m_\varphi(A) = < n^i, R^b >$ is a string of length $n^{bi} = n^k$. Thus the bit numbered (in n-ary) $j_1 j_2 \ldots j_{bi}$ is 1 iff $A \models \varphi(j_1, j_2, \ldots, j_{bi})$. If for all A in $STRUCT[\sigma]$,

$$A \in S \leftrightarrow m_\varphi(A) \in T$$

then m_φ is a k-ary *first-order interpretation* of S to T and we write $S \leq_{fo} T$ if such an interpretation exists. We refer the reader to [IL89] for further details and examples of such reductions.

Let $STRUCT[\tau]$ and $STRUCT[\sigma]$ be some vocabularies. A class C is closed under FO interpretations if for any problem $A \subset STRUCT[\tau]$ in C and for any problem $B \subset STRUCT[\sigma]$, $B \leq_{fo} A$ implies that B is in C.

Let P be the class of decision problems recognizable by deterministic Turing machines in time polynomial in the length of the input. To prove that SR contains P we will show that SR is closed under FO interpretations and that it contains a problem that is complete for P via FO interpretations.

Proposition 3.1 *SR is closed under FO interpretations.*

Proof: We have to show that if $A \in SR$ and $B \leq_{fo} A$ then B is in SR. This immediately follows from the observation that SR is closed under quantification and boolean operations. Closure under Boolean operations follows from the definition of

SRL. Closure under quantification is implicit in 2.1. Thus, for example to see that SR is closed under universal quantification, let $\varphi_1(\bar{x}, \bar{y}) be a function \in SRL$ and let $\varphi(\bar{z})$ be the FO formula $\forall y(\varphi_1(\bar{z}, \bar{y}))$ over the finite domain, D. Then, φ can be expressed in SRL as :
$\varphi(z_1, \ldots, z_c) =$ *set-reduce*$(D, \lambda(d,e)\varphi_1(e,d), \vee, \textit{true}, [z_1, \ldots, z_k])$
The existential quantifier case is handled similarly. ∎

Definition.

Let an alternating graph $G = (V, E, A)$ be a directed graph whose vertices are labeled universal or existential. Let $APATH(x, y)$ be the smallest relation on vertices of G such that

1. $APATH(x, x)$,
2. If x is existential (i.e. $\neg A(x)$) and for some edge (x, z) $APATH(z, y)$ holds then $APATH(x, y)$,
3. IF x is universal (i.e. $A(x)$) there is at least one edge leaving x and for all edges (x, z) $APATH(z, y)$ holds then $APATH(x, y)$.

Let $AGAP = \{G | APATH(V_0, V_{max})\}$.

The $\leq$ predicate, included in the base functions of SRL is crucial to the following result.

Fact 3.2 **([Imm87])** *$AGAP$ is complete for P under first-order reductions.*

Consider the following monotone operator Γ [Imm87]:-

$$\Gamma(R)[x, y] \equiv (x = y) \vee [(\exists z)(E(x, z) \wedge R(z, y)) \wedge (A(x) \rightarrow ((\exists z)E(x, z) \rightarrow R(z, y)))]$$

It is easy to see that $LFP(\Gamma) = APATH$. We show that it is possible to express $AGAP$ as a function in SRL in the following lemma:

Lemma 3.3 *$APATH$ is expressible in SRL.*

Proof: We shall specify the types in our SRL function for $APATH$ only at the beginning and then use variables without mentioning types to enhance the readability. We shall use Fact 2.1 extensively. Let $NODES$ of type $set(Vertex)$ and $EDGES$ of type $set([from, to : Vertex, label:\{AND, OR\}])$ be the input.

Thus the set of AND and OR labeled vertices can be obtained as follows:

$$ANDS = project(select(EDGES, \lambda(x)x.label = AND), from)$$

$$ORS = project(select(EDGES, \lambda(x)x.label = OR), from).$$

We can write Γ in SRL easily and then simulate the least fixed point operator on Γ which is of arity 2, by writing a loop which runs n^2 times.

$$\begin{aligned}\Gamma(x,y,R) = (x = y) \vee (&forsome(NODES, \lambda(z)(member([z,y],R) \wedge \\ &\qquad\qquad member([x,z], EDGES))) \\ &\wedge (\neg(member(x, ANDS)) \vee \\ &forall(NODES, \lambda(z)(\neg(member([x,z], EDGES)) \vee \\ &\qquad\qquad member([z,y],R)))))\end{aligned}$$

$$\begin{aligned}\Gamma_{x,y}(R) = set\text{-}reduce(&NODES, \\ &\lambda(d_1, S)(set\text{-}reduce(NODES, \lambda(d_2, e)([e, d_2]), \\ &\qquad \lambda(t,T)(if\ \neg(member([t.1, t.2], T)) \ \wedge\ \Gamma(t.1, t.2, T) \\ &\qquad\qquad then\ insert([t.1, t.2], T) \\ &\qquad\qquad else\ T), \\ &\qquad S, \\ &\qquad d_1)), \\ &union, \\ &R)\end{aligned}$$

$$\begin{aligned}&LFP_\Gamma = ITERATE() \text{ where} \\ &ITERATE() = set\text{-}reduce(NODES, identity, \\ &\qquad \lambda(d, Z)(set\text{-}reduce(NODES, identity, \lambda(d, X)\Gamma_{x,y}(X), Z)), \\ &\qquad \{\})\end{aligned}$$

∎

Corollary 3.4 $P \subseteq SR$.

Proof: Since $AGAP$ is complete for P under FO reductions (by Fact 3.2), and it is expressible in *SRL* (by Lemma 3.3), and *SRL* is closed under FO reductions (by Proposition 3.1), it follows that $P \subseteq SR$. ∎

Since we have defined *SRL* so that *set-height* is at most 1 and *tuple nesting and width* are constant, we have that

Proposition 3.5 *Let l be the tuple nesting and w be the tuple width of a tuple type α. Let S be of type* set of α *and let n be the number of elements in the input domain D. Then, $|S| \in O(n^w)$.*

Proof: The maximum size of any set S that can be formed is equal to the number of possible tuples of width w and nesting l which is easily seen to be $w^{l-1}n^w \in O(n^w)$. ∎

It now follows that

Lemma 3.6 $SR \subseteq P$.

Proof: Define depth, d, of a *set-reduce function* recursively:

$$
\begin{aligned}
&depth(base\ functions) = 0\\
&depth(set\text{-}reduce(S, appf, accf, base, e)) = 1 + max(depth(S), depth(appf), depth(accf),\\
&\qquad\qquad\qquad\qquad\qquad\qquad depth(base), depth(e))
\end{aligned}
$$

We show by induction on d that each function F in *SRL* can be computed in time polynomial in n and therefore produces sets of polynomial size.

Base case: $d = 0$. The base functions can clearly be computed in P. Only *insert* increases the set-size by 1.

Inductive Step: Any function in *SRL* is of the form $F(S, e) = set\text{-}reduce(S, appf, accf, base, e)$. By the inductive hypothesis, $accf, appf, base, e$ and S can be computed in time $\leq n^k$, for some constant k. Thus, we have $|S|$ applications of $appf, accf$ on inputs of size at most n^w by the proposition above. Total time to compute this recursion is

= the time to compute $accf, appf$ $|S| \leq n^w$ times
on input of size n^w + time to
compute *base* + the time to compute e
$\leq 2(n^w)(n^w)^k + n^k + n^k$
$\leq n^{k'}$, for some constant $k' = w(k+1) + 1$.

∎

Theorem 3.7 $P = SR$.

Proof: It follows from the Corollary 3.4 and Lemma 3.6 above. ∎

Remarks:

- It is possible to show that $DTIME(n^k) \subseteq SRL$ by directly simulating the Turing machine computation. Refer to section 6 where we give tighter bounds on the complexity of an *SRL* expression from its syntax.

- Let FP denote the class of functions computable in polynomial time. Then, it follows from the previous theorem that

  Corollary 3.8 (*Functions computed in SRL*) $= FP$.

- Restricting to a single usage of *set-reduce* does not help to restrict the complexity. It still remains sufficiently powerful to express $AGAP$ and hence the whole of P.

- The restriction on *set-height* is crucial as the following example shows. With *set-height* 2, it is possible to express a function in *SRL* that constructs a set of size exponential in the size of the input set.

  Example 3.9 *Consider the following function which given a set S constructs the power set $P(S)$ of S.*

  Finsert takes as arguments x, a two tuple of a set and an element, and a set of sets, T and returns $T \cup \{x.1\} \cup \{x.2 \textit{ inserted in } x.1\}$.

  $finsert(x, T) = insert(x.1, insert(insert(x.2, x.1), T))$

  Sift takes an element x and a set of sets T and calls *finsert* to insert x in each one of the elements of T and returns T unioned with all these new sets containing x.

  $sift(x, T) = \textit{set-reduce}(T, \lambda(y, e)([y, e]), finsert, \{\}, x)$

  $powerset(S) = \textit{set-reduce}(S, identity, sift, \{\{\}\})$

  Similarly, it can be shown that such a situation exists if we do not restrict the *tuple nesting*. In particular, regard T in the program above as a set of tuples of width 2 and arbitrary nesting, and redefine *finsert* as follows

$finsert(x, T) = insert(x.1, insert([x.2, x.1], T))$

$Powerset(S)$ is now *set-reduce*$(S, identity, sift, \{[-,-]\})$

In [SS89], a *list-reduce* construct is defined which is exactly the same as *set-reduce* except that the object we recurse over is a *list*, and *not* a *set*. The difference is that the items appear in a specific order in the list. Clearly any function realized using *set-reduce* can be implemented using *list-reduce* by simply substituting the former by the latter construct. Define *list-height* analogous to *set-height*. Let us denote the problems expressed by the corresponding language with *list-height* ≤ 1 as LR. As observed above, $SR \subseteq LR$. But $LR \not\subseteq P$. This can be seen from the following function which is not in P, but is in LRL viz. $F(<1>, <1, 2, .., n>) = <1, 1, \ldots, (2^n\ times), \ldots, 1>$. Note that lists can be of arbitrary length in this language. In fact, we will see that LR exactly equals the class of *primitive recursive sets*.

The proof of Lemma 3.6 goes through as long as Proposition 3.5 is not violated. So what are the operators that can be included in *set-reduce language* such that it still remains within P? Clearly integers and bounded operators on them such as *addition mod* x and *multiplication mod* x, where x is an exponential function of n (the input size), can be added to *set-reduce language* without blowing up its complexity, since the size of such objects is bounded by $\log x$, which is polynomial in n. Let $\mathbf{N}$ denote the set of natural numbers. Let *succ* denote the successor operator on the naturals. It is shown later that allowing objects of type N or integers, with the usual *succ* operator on such types, in *set-reduce language* increases the complexity to that of *primitive recursive functions*. In particular, if we allow addition or multiplication on integers and allow the type, *set of integer*, then we can express the class of primitive recursive functions in this language.

However, if we do not permit the use of the type, *set of* $\mathbf{N}$ or *set of integer*, then its complexity is still within P. For example, we can safely add integer types along with addition ($+$) on integers in *set-reduce language* while still remaining within P provided we do not allow the type, *set of integer*. We can also add the operation of multiplication ($*$) on integers, if in addition to the previous restriction, we do not allow the accumulator function, $accf$, to use it. Clearly, addition and multiplication are in P and hence, their result is polynomial sized.

Let x and y be of some ordinal type α, like N. Let $< op >$ be some operator such that if $x\ < op >\ y$ is repeated n times recursively then size of the result is polynomial in the sizes of x and y, and in n. Let $a = x\ op\ y$ and let $|x|$ denote the size or length of the binary representation of x. SRL is closed with respect to such an operator $< op >$ defined on α, provided the type - *set of* α - is not permitted. An example would be some $< op >$ such that $|x\ < op >\ y|$ is an additive function of $|x|$ and

$|y|$. But if $|x \ < op > \ y|$ were a multiplicative function of $|x|$ and $|y|$, then we have to impose one further restriction - we do not allow *accf* in the set-reduce construct to use $< op >$ unless one of the operands is a fixed constant. If *op* is $+$ then $|a| \leq max\{|x|, |y|\} + 1$, whereas if *op* is $*$ then $|a| \leq 2 * max\{|x|, |y|\}$. Thus, if one allows *accf* to use multiplication, then it is easy to compute x^{2^n} (which cannot be done in P) in *set-reduce language* by repeated squaring. However, we can allow *multiplication by a constant* inside *accf*, while still remaining within P, since in this case, the size of the result of n repeated multiplications by a *constant* is clearly polynomially (in fact linearly) bounded. It can be easily observed that

Proposition 3.10 *Addition of other operators and functions to SRL will not take us out of P provided that the size of the sets we can build, using those operators, remains polynomially bounded.*

4. Restricted versions of SRL

Let L (NL) denote the class of problems recognized by deterministic (non-deterministic) Turing machines using space no more than logarithmic in the input size. It is well known that $L \subseteq NL \subseteq NC^2$. The question arises as to whether there exist any syntactic restrictions on *SRL* that in an elegant and natural way capture L and NL. Characterizing L and NL as some form of SRL would be interesting since problems in these classes are also efficiently parallelizable.

One way of doing this follows easily from the results in [Imm87]. We adopt the same notations. Let $\varphi(\bar{x}, \bar{x'})$ be any FO formula. Define $TC[\lambda\bar{x}, \bar{x'}\varphi]$ as the reflexive, transitive closure of the relation φ. Let $(FO + TC)$ be the set of properties expressible using first order logic plus the operator TC. The following characterization is well known:

Fact 4.1 ([Imm87, Imm88]) $NL = (FO + TC)$.

We define a new operator called TC, in *SRL* as follows. Let the set of vertices be D. $TC(\varphi)$ is computed as follows in *SRL*:

Let $EDGEp([x,y]) = \varphi(x,y)$, and $EDGES = select(join(D, D), \lambda([x,y])EDGEp([x,y]))$

The function $bothsides(v, EDGES)$ forms the set of tuples of nodes entering v and those leaving v:

$bothsides(v, EDGES) = join(D, D, \lambda(t1, t2)((t1.to = v) \vee (t2.from = v)), \lambda(t1, t2)([t1.from, t2.to]))$
$addedges(v, E) = union(E, bothsides(v, E))$

Finally,
$TC(EDGES) = \textit{set-reduce}(\textit{project}(EDGES, to), \lambda(x)(x), \lambda(x,Y)(\textit{addedges}(x,Y)), EDGES)$

Let $SRFO + TC$ be the class of problems expressible in a subset of SRL that has only the following functions available: $\textit{forsome}, \textit{forall}, \neg, \vee, \wedge, \leq, TC$. As an immediate corollary to the preceding fact, we have that

Corollary 4.2 $SRFO + TC = NL$.

Proof: Clearly every property expressible in $FO + TC$ can be expressed in $SRFO + TC$ and vice versa. ∎

Given a first order relation $\varphi(\bar{x}, \bar{x'})$, let

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

That is, $\varphi_d(\bar{x}, \bar{x'})$ is true just if $\bar{x'}$ is the unique descendant of $\bar{x}$. Define $DTC(\varphi) \equiv TC(\varphi_d)$. Let $(FO + DTC)$ be the set of properties expressible using first order logic plus the operator DTC. Then, analogous to the NL case, it comes as no surprise that,

Fact 4.3 **([Imm87])** $L = (FO + DTC)$.

$DTC(\varphi)$ can be computed in SRL as follows:

$$\varphi_d(\bar{x}, \bar{y}) = \varphi(\bar{x}, \bar{y}) \wedge \ \textit{forall}(D, \lambda(z,e)(p(z,e)), [\bar{x}, \bar{y}])$$

$$\textit{where } p(\bar{z}, e) = \neg(\varphi(e.1, \bar{z})) \vee (\textit{equal}(e.2, \bar{z}))$$

$$DTC(\varphi) = TC(\varphi_d).$$

Let $SRFO + DTC$ be the class of problems expressible in a subset of SRL that has only the following functions available: $\textit{forsome}, \textit{forall}, \neg, \vee, \wedge, \leq, DTC$. Thus, we have the following easy corollary from Fact 4.3 that

Corollary 4.4 $L = SRFO + DTC$.

Another, perhaps more natural, way of characterizing L is achieved by considering the following restriction on SRL : we restrict the function acc in our *set-reduce* template to return just a tuple of bounded width (and set-height zero). Let us denote this version of SRL as $BASRL$ and the set of properties expressible in this version of SRL as $BASR$. Then, we can show that the class L is exactly equal to $BASR$ as follows. The proof is similar in form to that of P equals SR. We need the following definitions.

Let $BIT(i,x)$ denote the value of the i^{th} bit in the binary representation of x. In the context of SRL, since it only deals with sets and not numbers, we have to impart a meaningful interpretation to this operator. Assume the active domain of any SRL program is D and let $|D|$ denote the size of D and let $n = |D|$.

Note that we have a total order $\leq$ on D which is the order in which the elements of D are scanned by *set-reduce*. We can either assume that it is available to us as a set of pairs say, $S = \{(a,b)|a \leq b\}$, or we can compute the successor or predecessor of an element with respect to $\leq$ whenever we need it. Each element has a unique position in this ordering. Let d_1, d_2 be any elements in D, let i_1, i_2 be the ranks (positions) of d_1, d_2 in that total order. Then, $BIT(d_1, d_2) \equiv BIT(i_1, i_2)$. In a similar vein we define addition, multiplication, exponentiation. Let $d_1, d_2 \in D$ and let i_1, i_2 be their respective ranks in the ordering $\leq$. Then $d_1 + d_2$ is defined to be $d_3 \in D$ such that if i_3 is the rank of d_3 in $\leq$ then $i_3 = i_1 + i_2$. Multiplication and exponentiation are likewise defined.

Proposition 4.5 *Addition, multiplication, exponentiation are expressible in BASRL.*

Proof: We show how to add 1 as follows:

$$\begin{aligned}
increment(a) = \textit{set-reduce}\ (&D, identity,\\
&\lambda(d,X)(if\ \neg(X.1) \wedge (d = X.3)\\
&\qquad then\ [true, false, X.3]\\
&\qquad else\ if\ \neg(X.2) \wedge (X.1)\ then\ [X.1, true, d]\\
&\qquad else\ X),\\
&[false, false, a])
\end{aligned}$$

Similarly one can define *decrement*(A). We have to take care of the boundary cases - *increment*(n) and *decrement*(0) appropriately. Then,

$$\begin{aligned}
ADD(a,b) = \textit{set-reduce}(&D, identity,\\
&\lambda(d,X)(if\ \neg(X.1 = n)) \wedge \neg(X.2 = 0)\\
&\qquad then\ [increment(X.1).2, decrement(X.2).2]\\
&\qquad else\ if\ (X.2 = 0)\ then\ X\\
&\qquad else\ [0, decrement(X.2).2]),\\
&[a,b])
\end{aligned}$$

Note that ADD, *increment*, *decrement* all return a tuple ($[-,-]$) and operators .1 and .2 return the first and second component of the tuple respectively. Multiplication is expressed as follows:

$$
\begin{aligned}
MULT(a,b) = \textit{set-reduce}(&D, \lambda(s, extra)(extra),\\
&\lambda(e,X)(if\ (X.2=0)\ then\ X\\
&\qquad\qquad else\ [ADD(e,X.1).1, decrement(X.2).2]),\\
&[0,b],\\
&a)
\end{aligned}
$$

Note that we use 0, n to simply mean the first and last elements respectively in $\leq$. Hence, $x = 0$ or, $x = n$ can easily be checked in $BASRL$ by seeing whether x is the first or, last element of the ordering.
Exponentiation is expressed as below:

$$
\begin{aligned}
EXP(a,b) = \textit{set-reduce}(&D, \lambda(s,x)(x),\\
&\lambda(x,T)(if\ (T.2=0)\ then\ T\\
&\qquad\qquad else\ [MULT(x,T.1).1, decrement(T.2).2]),\\
&[1,b],\\
&a)\ \blacksquare
\end{aligned}
$$

Lemma 4.6 *BIT is expressible in $BASRL$.*

Proof: We shall use the proposition above. First we show how to divide by 2 in $BASRL$:

$$
\begin{aligned}
SHIFT(a) = \textit{set-reduce}(&D, identity,\\
&\lambda(x,e)(if\ \neg(e.1) \wedge ((ADD(x,x).1) = e.2)\\
&\qquad then\ [true, x, false]\\
&\qquad else\ if\ (increment(ADD(x,x).1).2 = e.2)\\
&\qquad then\ [true, x, true]\\
&\qquad else\ e),\\
&[false, a, false])
\end{aligned}
$$

Note that we have also defined a predicate, $PARITY$ of a number as *true iff number is odd*:

$$PARITY(x) = SHIFT(x).3$$

Finally, $BIT(i,a)$, i.e. the i^{th} bit of a is given by the parity of a divided by 2^i as follows:

$$\begin{aligned}
REM(i,a) = \textit{set-reduce}(&D, \textit{identity},\\
&\lambda(s,X)(\textit{if}\ \neg(X.1=0)\\
&\qquad \textit{then}\ [\textit{decrement}(X.1).2, SHIFT(X.2).2]\\
&\qquad \textit{else}\ X),\\
&[i,a])
\end{aligned}$$

$$BIT(i,a) = PARITY(REM(i,a).2)$$

∎

Corollary 4.7 *$BASRL$ is closed with respect to FO interpretations that also use BIT.*

Proof: Let *struct*(σ) and *struct*(τ) be some vocabularies. Let $A \subset$ *struct*(σ) and $B \subset$ *struct*(τ) be two problems. Given that $A \in BASRL$ and $B \leq_{fo+bit} A$ we have to show that $B \in BASRL$. It follows immediately from 3.1 that $BASRL$ is closed with respect to quantification and boolean operations, since the *set-reduce* functions defined in that proof satisfy the definition of $BASRL$. Closure under BIT operation i.e. for any function f, $f \in BASRL \rightarrow BIT(f,i) \in BASRL$, follows from Lemma 4.6 above. Note that f returns a singleton element from the active domain which is handled by the lemma, or it returns a bounded width tuple of elements, in which case, BIT is interpreted with respect to the ordering on the tuple induced by $\leq$. It is a straightforward but tedious exercise to extend Lemma 4.6 to this case. ∎

Let S_n denote the group of permutations on $1, 2, \ldots, n$ under composition. Let IM_{S_n} denote the following iterated multiplication problem: given permutations $s_1, \ldots, s_n \in S_n$ as input, compute their composition, i.e. $s_1 * s_2 * \cdots * s_n$. The following theorem indicates the usefulness of IM_{S_n}.

Fact 4.8 (**[CM87, IL89]**) *IM_{S_n} is complete for L under FO reductions with BIT.*

We show how to express IM_{S_n} in $BASRL$.

Lemma 4.9 *IM_{S_n} is expressible in $BASRL$.*

Proof: We shall express IM_{S_n} as a predicate in the sense that given the input as stated earlier and also two other inputs viz. numbers i and j, our $BASRL$ program will return true iff the iterated product permutation maps i to j. The input will be coded as follows: each permutation would be represented by tuples of the type, $[i, [j, k]]$, which means that the i^{th} permutation maps j to k. Thus, the input,say I, is a set of such tuples. Note that since i, j, k are represented by sets of respective cardinalities the input is of *set-height* 2. It is easy to write a program in $BASRL$ to check that the permutation group is indeed S_n, where n is the number of elements (permutations) being multiplied. Also, n can be regarded as a constant available to us since one can always define it in FO as follows:

$$\exists x \forall y (y \leq x).$$

Then, the following program expresses IM_{S_n}. As before, we do not specify the types in the following to make it easy to read.

$$
\begin{aligned}
IP(I,i) = \textit{set-reduce}(I, \textit{identity},\ & \\
& \lambda(xtuple, pair)(\textit{set-reduce}(I, \textit{identity}, \\
& \qquad \lambda(x,p)(if\ (x.1 = p.1)\ \wedge (x.2.1 = p.2) \wedge \neg(p.1 = n) \\
& \qquad \qquad then\ [increment(p.1).2, x.2.2] \\
& \qquad \qquad else\ p), \\
& \qquad pair) \\
& [1, i])
\end{aligned}
$$

$$IM(I,i,j) = if\ IP(I,i).2 = j\ then\ true\ else\ false$$

Note that the accumulator function returns a bounded tuple in the above program. ∎

Corollary 4.10 $L \subseteq BASR$.

Proof: Since IM_{S_n} is complete for L under *FO interpretations* that include BIT (by Fact 4.8), and it is in $BASRL$ (by Lemma 4.9), and $BASRL$ is closed under these reductions (by Corollary 4.7), the result follows. ∎

Lemma 4.11 $BASR \subseteq L$.

Proof: It suffices to show that a logspace deterministic Turing machine can simulate any $BASRL$ program. Since the accumulator function only returns a bounded width tuple, we can just write the tuple on $O(\log n)$ bits of worktape. It is easy to see that the scan done by the set-reduce can be simulated by just scanning the input with the read-only head and an index tape that uses at most $\log n$ bits. Now all that remains is to show the closure under bounded number of compositions. This follows from the well known fact that logspace computable 0-1 functions are closed under compositions. ∎

Finally, we have that,

Theorem 4.12 $L = BASR$.

Proof: It follows from the previous lemma and the corollary. ∎

Remarks. $BASRL$ programs can be evaluated efficiently in parallel (since, $L \subseteq IND(\log n) \subseteq NC^2$).

5. Expressiveness of unrestricted SRL

SRL, without any restrictions, promptly becomes intractable. Define *unrestricted SRL* as *SRL* without the restrictions on set-height and tuple nesting and width. Equivalently, we can restrict set height and throw in an additional operator, *new*, that gives the language in effect the ability to construct a new element. In particular, let $new(D)$ return an element $\notin D$, where D is any set. Note that this is equivalent to having an unbounded successor operator. Let $newSRL$ denote this version of *SRL*. At first glance it seems that these versions of *SRL* are not that different from *SRL*. However, we show that these versions express all the primitive recursive functions. Observe that *SRL* contains no successor function whereas *unrestricted SRL* and *new SRL* both contain a successor function.

Let *PrimRec* denote the class of primitive recursive functions. The latter usually map $\mathbf{N} \rightarrow \mathbf{N}$, where $\mathbf{N}$ is the set of natural numbers.

Let us recall the definition of *PrimRec* [DW]. Let $g : \mathbf{N} \rightarrow \mathbf{N}$, $h : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$. Then, $f : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ is defined by primitive recursion from g, h if

$$f(0, t) = g(t)$$

$$f(s + 1, t) = h(s, g(s, t))$$

Consider the so-called *initial* functions :

$$succ(i) = i + 1$$

$$n(i) = 0$$

$$p_k^n([i_1, \ldots, i_n]) = i_k$$

A function is *primitive-recursive* if it is obtained from the *initial functions* by a bounded number of compositions and primitive-recursions.

Note that functions in *newSRL* (or *unrestricted SRL*) give mappings between sets. However, we can consider them as functions from **N** to **N**, since finite ordered sets can be Gödel numbered in a standard way.

In the *unrestricted SRL* case, we use Von Neumann's mapping between sets and natural numbers

$0 = \varphi$, $1 = \{\varphi\}$, $2 = \{\varphi, \{\varphi\}\}$, $3 = \{\varphi, \{\varphi\}, \{\varphi, \{\varphi\}\}\}, \cdots$, $n + 1 = n \cup \{n\}$, ...

It is clear that $n - 1$ and $n + 1$ can easily be expressed in *unrestricted SRL* using this representation for n. Note that the *set-height* grows in an unbounded manner.

In the *newSRL* case, we allow a *new* operator that gives us a new element, and in this notation,

$0 = \varphi, 1 = \{d_1\}, 2 = \{d_1, d_2\}, \ldots, n + 1 = \{d_1, \ldots, d_n, new\}$ ....

In this case, the set-height does not grow. It is easy to express $i + 1$ in *newSRL*. In the following, *unrestricted SRL* and *newSRL* denote the functions from **N** to **N** that can be expressed in the corresponding languages.

Theorem 5.1 *PrimRec* = unrestricted *SRL* = *newSRL*.

Proof: The initial functions are easily expressible in *unrestricted SRL* (*newSRL*). Eg.,

$$proj_k(t) = t.k$$

$$succ(S) = union(S, insert(S, \{\}))$$

$$(succ(S) = insert(new(S), S)).$$

In fact, this is the only usage of *new*.

(i):*PrimRec* $\subseteq$ *Unrestricted SRL* (*newSRL*):

It follows easily from the following

Proposition 5.2 Unrestricted *SRL (newSRL) is closed with respect to primitive recursion.*

Proof: Let f be the function obtained from g, h by primitive recursion as defined above. We show how to compute f in *unrestricted SRL* or *newSRL*.

$f(S,T) =$
$\textit{set-reduce}(S, \lambda(x)(x), \lambda(x,T')(hf(x,T')), [g(T), \{\}])$
where $hf(x,T') = [h(T'.2, T'.1), insert(x, T'.2)]$ ∎

(ii):*unrestricted SRL* $(newSRL) \subseteq PrimRec$:

Let us encode the ordered sets used by the SRL expression by their Gödel numbers. We show how to simulate the *set-reduce* operator using primitive recursion and since *PrimRec* and *SRL* are closed under composition and recursion, the result follows. The base functions in *SRL* are clearly primitive recursive. Note that the order in which *set-reduce* scans a set is given by the base function $\leq$. Let $accf$, $appf$, $base$ be primitive recursive functions. Then any *set-reduce* expression in *SRL*, e.g.

$$f(S,y) = \textit{set-reduce}(S, appf, accf, base, y)$$

is equivalent to the following primitive recursive function:

$$\begin{aligned} f(enc(0), enc(y)) &= base(enc(y)) \\ f(enc(s)^{+}, enc(y)) &= accf(appf(enc(s), enc(y)), f(enc(s), enc(y))) \end{aligned}$$

where $enc(s), enc(s)^{+} \in S$, $enc(x)$ means encoding of x and $enc(s) \leq enc(s)^{+}$). Note that for numbers s^{+} such that s is not the encoding of any element in S, $accf(x,y) = y$ i.e. it simply ignores them. ∎

Remarks:

- It can be shown in a manner similar to the proof above that

  Corollary 5.3 $LR = PrimRec$.

- Note the crucial use of the types, N and *set of* N, in $newSRL$ in the context of the comments preceding 3.10. Thus, we see that merely throwing in *new* operator increases the complexity of SRL all the way to $PrimRec$.

6. Complexity of SRL from its syntax

Given a program in *set-reduce language*, and the results in this paper, a scan of its syntax allows us to make certain conclusions regarding its complexity. If the user has sets of *set-height* greater than 1 in the program, then its complexity may be exponential. On the other hand, if sets of *set-height* at most 1 are used, then its complexity is polynomial in the size of the input sets. If in addition, the accumulator functions, $accf : \{\alpha, \gamma\} \rightarrow \beta$, (for some types α, γ, β) in his *set-reduce* constructs are such that β for any $accf$ is never of type *set*, then we are certain that the function expressed by the program is in L (or logspace). Any usage of the type *set of* some *unbounded type* in the program would possibly make the function it is computing very hard to optimize, but on the other hand using objects of unbounded type without using *set of* such objects makes it less difficult.

Let a be the maximum width of a SRL expression, i.e. the maximum arity of tuples used in a non-input set. Let d be the depth (defined in Lemma 3.6) of the expression. Let T_{ins} be the time complexity of an *insert* operation. Let n be the size of the input. Keeping in mind that the sets dealt with are of size polynomial in n, T_{ins} could be $O(1)$ (implemented by hashing), $O(\log n)$ (implemented by some balanced data structure) or at worst n^a, the maximum size of any set in the expression. Let $DTIME(f(n))$ denote the class of problems recognized by deterministic Turing machines in time bounded by $O(f(n))$. Then, we can easily bound the time complexity as follows

Proposition 6.1 *Any SRL expression with width a and depth d is in* $DTIME(n^{ad}T_{ins})$.

Proof: By induction on the depth d.
$d = 0$: The base function *insert* takes time T_{ins}.

Any *set-reduce* over a set, say R, of depth d, where the *accf* and *app* functions are themselves of depth $d-1$, takes time
$\leq |R| \ (max\{time\ of\ the\ accf\ or\ app\})$
$\leq n^a n^{a(d-1)} T_{ins}$ by the ind. hyp.
$= O(n^{ad}T_{ins})$ ∎

The bound leaves much room for improvement. In actually analysing a particular SRL expression, one usually can do much better. since then one can get rid of the overestimated n^a term that appears in the proposition above. Is $DTIME(n)$ expressible by a SRL expression with width 1 and depth 1? Apparently not. We show in the following that $DTIME(n)$ can be expressed by a SRL expression of width 2 and depth 2. However, the expression we obtain can actually be evaluated in time $O(n^2T_{ins})$ which is much better than the bound $O(n^4T_{ins})$ given by 6.1 above.

Proposition 6.2 *$DTIME(n)$ is expressible by a SRL expression of width* 2 *and depth* 4.

Proof: We show how to simulate the computation of a $DTIME(n)$ Turing machine by an SRL expression. Let σ be the alphabet, $x_1, \ldots, x_n$ be the input where $x_1, \ldots, x_n \in \sigma$, and n be the input size. Let S denote the input as a set of pairs viz. $\{[1, x_1], [2, x_2], \ldots, [n, x_n]\}$. Let us denote the work tape W as another set of pairs. It is easy to write a SRL expression, call it *create-tape*, that initializes W with blanks i.e. $\{[1, -], [2, -], \ldots, [n, -]\}$. Let us denote the input tape head and work tape head positions by two variables, say P_1, P_2. Now we can just use a *set-reduce* over S, thereby iterating n times, and in each iteration the *accf*, in this case, $F1$, updates W,P_1,P_2 according to the Turing machine program:

$set\text{-}reduce(S, identity, \lambda(s, T)F1(T), [W, P_1, P_2])$

$$F1(T) = set - reduce(S, identity, \lambda(s, X)F2(s, X), T)$$

$F2(s, X) = if\ (s.1 = X.2)$
$\quad then\ set\text{-}reduce(X.1, \lambda(t, ex)[t, ex], \lambda(tE, Y)update(tE, Y), [\{\}, 0, 0], [X.2, X.3, s.2])$
$\quad else\ X$

$Update(tE, R) = (if\ (tE.1.1 = tE.2.2)$
$\quad then\ use\ TM\ transition\ table\ and\ tE.1.2$
$\quad\quad (work\ tape\ content)\ and\ tE.2.3(input\ cell)\ to\ make$
$\quad\quad a\ move\ i.e.\ change\ work\ head\ position\ tE.2.2$
$\quad\quad and\ input\ head\ position\ tE.2.1\ accordingly\ and$
$\quad\quad return\ [insert([tE.1.1, tE.1.2'], R.1), tE.2.1', tE.2.2']$
$\quad else\ [insert(tE.1, R.1), R.2, R.3]$

Note that W is a set of pairs and hence the width is 2. $F1$ is of depth 3, since it uses one *set-reduce* over S to get the input tape content and another over W, which itself is given by a depth 1 *set-reduce* function called *create-tape*, to get at the work tape content. The total depth equals 4. ∎

Note that we use the *increment* function implicitly in updating the head positions and that that the *set-reduce* over W is repeated only once for one full scan of S, and *increment* is also done only once for one full scan of W. An analysis of the time complexity of this expression reveals that the two *set-reduce*'s in $F1$ together take $O(nT_{ins})$ time and since it is iterated over n times, the total complexity is n^2T_{ins}.

The proof above can easily be generalized to show that

Corollary 6.3 *$DTIME(n^k)$ is expressible by a SRL expression of width $k+1$ and depth $k+4$.*

Remarks. The SRL expression obtained above can be evaluated in time $O(n^{2k}T_{ins})$.

Let SR_h denote the class of problems expressible by a version of *set-reduce language* that has its *set-height* h and *tuple-width* $\leq k$, for some constant k. Let n denote the input size. Let $2^{i\#n}$ denote a stack of i 2's, i.e.$2^{0\#n} = n^k$, $2^{i+1\#n} = 2^{2^{i\#n}}$.

Then, following the preceding proof, it can be shown that

Corollary 6.4 *For $h = 1, 2, \ldots$ $SRL_h = DTIME(2^{h\#n})$.*

Remarks. This hierarchy is mentioned here for the sake of completeness. It is quite similar in notion to the results of [HS88], [AV88] and others.

7. The Rôle of Ordering

A set stored by a computer has its members in some order. Simply put, any object is a sequence of bits, thus falling in place in lexicographical order. This allows any database system to search through a set in lexicographical order à la *set-reduce*; and, also to compute information that may depend on the somewhat arbitrary ordering that ensues. For example, one may compute the order dependent boolean query:

$$\text{Purple}(\text{First}(S))$$

namely that the element that happens to be first in the arbitrary ordering of the set S satisfies the predicate Purple($\cdot$).

It is neither surprising, nor especially dangerous that programs that search through a set in a given order may compute some information that depends on that order. If the order is truly independent of any information we wish to be computing and if our programs are correct, then the answers will be independent of the ordering. Furthermore, most sets of data have at least one natural ordering which can be used instead of the arbitrary ordering, for example one can print the elements of a set of employees in order of their names, or, date of hire, etc.

Still, if we are not certain that our programs are correct, then it would be nice to know whether the answers we get depend on the arbitrary ordering of elements within a set. Furthermore, one can imagine difficulties when long queries are suspended and then resume, or when different parts of them are carried out at different sites of a distributed database. In particular, these separated processes may be using different, arbitrary ordering of the same set in which case, just combining their computations without taking note of their dependence on the ordering, could lead to error.

In any case, there is general sentiment in the theoretical database community that ordering is dangerous and that order dependent queries should be avoided. In fact, in the influential paper [CH82a], Chandra and Harel define a *query* to be an order-independent query and they ask the question:

Question 7.1 *Is there a* natural *language that expresses exactly the set of polynomial-time computable, order-independent queries?*

One can make this question more precise by removing the undefined term "natural" and instead ask:

Question 7.2 *Is there a recursively enumerable set of programs that compute exactly the set of polynomial-time computable, order-independent queries over relational databases?*

The above two questions remain open in spite of many years of intensive study. See [IL90] for a history of this subject. Here we give an overview of what is known about Questions 7.1 and 7.2.

In a preliminary version of the paper [CH82a], Chandra and Harel defined fixed point logic, FP, which is an extension of first-order logic to include applications of the fixed point operator, thus allowing the inductive definition of new relations. In symbols: FP = (FO(wo$\leq$)+ LFP). Chandra and Harel conjectured that there was a hierarchy of queries in FP consisting of successive applications of LFP and first-order operations. In response, Immerman showed that Chandra and Harel's conjecture was false:

Fact 7.3 (**[Imm82, Imm87]**) *Every query in FP is expressible the form*

$$\mathrm{LFP}(\varphi(R)[\bar{t}]$$

where $\bar{t}$ is a tuple of terms and φ is a quantifier-free formula containing no occurrences of LFP.

Perhaps more interesting is the fact that if a total ordering of the universe is present, then the queries expressible in (FO + LFP) are exactly those computable in polynomial time.

Fact 7.4 ([Imm82, Va82])

$$(\mathrm{FO} + \mathrm{LFP}) \quad = \quad \mathrm{P}$$

Fact 7.4 fails badly if we remove the ordering. For example, it is easy to show that without an ordering we cannot count. In fact, if EVEN represents the query that is true if the size of the universe is even, then:

Fact 7.5 ([CH82a]) *EVEN is not expressible in* (FO(wo$\leq$)) + LFP).

Indeed, before 1989, examples involving the counting of large, unstructured sets were the only problems known to be in order-independent P but not in (FO(wo$\leq$) + LFP). In 1982, Immerman [Imm82] considered the language (FO(wo$\leq$)+LFP+count) in which structures are two-sorted, with an unordered domain $D = \{d_0, d_1, \ldots, d_{n-1}\}$ and a separate number domain: $N = \{0, 1, \ldots, n-1\}$ with the database predicates defined on D and the standard ordering defined on N. The two sorts are combined via counting quantifiers:

$$(\exists i\, x)\varphi(x)$$

meaning that there exist at least i elements x such that $\varphi(x)$. Here i is a number variable and x is a domain variable.

For quite a while, it was an open question whether the language (FO(wo$\leq$)+ LFP + count) is equal to order independent P. A positive answer would have provided a nice solution to Question 7.1.

Instead, in [CFI89] it was proved that that (FO(wo$\leq$) + LFP + count) is strictly contained in order-independent P. See Theorem 7.8 below for an explanation and slight generalization of this result.

See Figure 7.6 for the relationships between the polynomial-time query classes we have been discussing.

$$\begin{aligned} (\mathrm{FO}(\mathrm{wo}\leq) + \mathrm{LFP}) &\subset (\mathrm{FO}(\mathrm{wo}\leq) + \mathrm{LFP} + \mathrm{count}) \\ &\subset (\text{order-independent P}) \\ &\subset (\mathrm{FO} + \mathrm{LFP}) = \mathrm{P} \end{aligned}$$

Figure 7.6: Some polynomial-time query classes.(The relation "$\subset$" denotes proper containment.)

Another attempt to capture the polynomial-time, order-independent queries is worthy of mention here. In [OBB89] the language Machiavelli is defined. It contains an operator called hom which is similar to set-reduce. In the following definition, op is any previously defined binary operation.

$$\begin{aligned}\text{hom}(f, \text{op}, z, \{\}) &= z \\ \text{hom}(f, \text{op}, z, \{x_1, x_2, \ldots, x_n\}) &= \text{op}(f(x_1), \ldots, \text{op}(f(x_n, z)) \ldots))\end{aligned}$$

It is not hard to see that in the presence of an ordering, and with set-height restricted to at most one, the languages SRL and a similar *hom*-based language, which we will refer to as HL, have equivalent expressive power. However, in [OBB89], an instance of hom is called *proper* if the corresponding op is commutative and associative. It follows that an application of proper hom does not derive any information from the ordering in which a set is presented. Thus the language "proper HL" is order-independent and would seem to be a candidate for order-independent P.

One obstacle to this is easily overcome: when op is associative, the application of hom may be drawn as a binary tree of height $\log n$, and thus evaluated in parallel time $O[\log n]$ times the parallel time to perform a single op. It follows that "proper Machiavelli" is contained in the class NC consisting of those problems computable in parallel time $(\log n)^{O[1]}$ using polynomially many processors. NC is believed to be strictly contained in P [C85].

We can alleviate this problem by allowing "proper HL" to iterate an operation polynomially many times. One way to do this is to consider the language similar to (FO(wo$\leq$) + LFP + count) which has a number domain, N, separate from the database domain. One can then safely allow arbitrary applications of hom over the number domain. Define (FO(wo$\leq$) + N + hom) to be this class. Then we have the following proposition which says that "proper HL together with a polynomial iteration operation" is at least as expressive as (FO(wo$\leq$) + LFP + count). As of this writing, we do not know whether or not this inclusion is proper:

Proposition 7.7

$$(\text{FO}(\text{wo}\leq) + \text{LFP} + \textit{count}) \subseteq (\text{FO}(\text{wo}\leq) + N + \textit{hom})$$

Proof The above discussion explained why (FO(wo$\leq$)+N+hom) contains (FO(wo$\leq$)+ N + LFP). Thus it suffices to show how to count using proper hom. This is easy.

Let $f : D \rightarrow N$ be the function that takes everything in the database domain to the number 1. Then we can count a set $S \subseteq D$ using hom as follows:

$$\text{count}(S) = hom(f, +, 0, S)$$

∎

We next show that the lower bound from [CFI89] **does** apply to the language $(\text{FO}(\text{wo}\leq) + N + \text{hom})$. It also applies to the language $(\text{FO}(\text{wo}\leq) + \text{count} + \text{while})$.[2]

Theorem 7.8 *The set (order-independent* P*) is not contained in* $(\text{FO}(\text{wo}\leq) + N + hom + while)$.

Proof The paper [CFI89] constructs a sequence of structures G_n, H_n, $n = 1, 2, \ldots$. These structures contain $O[n]$ domain elements. G_n and H_n may be distinguished in linear time if we have access to any ordering on their domains. By contrast, G_n and H_n agree on all sentences in $(\text{FO}(\text{wo}\leq) + \text{count})$ containing at most n distinct variables. (If the simple, polynomial-time order-independent property that characterizes G_n were expressible in $(\text{FO}(\text{wo}\leq) + \text{LFP} + \text{count})$ or in $(\text{FO}(\text{wo}\leq) + \text{count} + while)$ then it would follow that a first-order sentence with a *bounded* number of variables would distinguish the graphs G_n and H_n. This is true because the operators LFP and 'while' are simply "formula iterators" and do not increase the number of distinct variables in the formula.)

Now, we show that over the structures G_n, H_n applications of hom give us no new expressive power. This is because G_n and H_n are almost ordered. That is, there is a first-order, quasi-total ordering on the vertices. The vertices are partitioned into color classes of size at most 4 and the color classes are totally ordered. Thus we can compute hom of a set by walking through the color classes occurring in the set, applying the operator by hand to at most four elements in each class. ∎

One of us (Immerman) has studied the issue of ordering because of its intimate connection with his study of descriptive and computational complexity [IL90]. Another of us (Stemple) has developed a theory of finite sets because of their importance in database transactions [SS89]. It is an unaesthetic aspect of any such theory to date.

[2] In [Va82], Vardi defined the language (FO + while), i.e. first-order logic together with an unbounded iteration operator, and showed that its expressive power is equal to PSPACE. (See also [Imm82b] for an equivalent formulation of an unbounded iterator applied to FO giving PSPACE.) See also [AV91] for a surprising new result: $(\text{FO}(\text{wo}\leq) + \text{while}) = (\text{FO}(\text{wo}\leq) + \text{LFP})$ if and only if P = PSPACE.

that in order to develop a theory of unordered finite sets that is rich enough to describe computation, one seems to need an ordering on these sets.

It seems to us unacceptable to use impoverished query and transaction languages in order to have the aesthetically desirable characteristic of order-independence. Our view is that one should use a language that we know includes all the feasible queries, i.e. P. But, that one should use a theorem prover such as Sheard's extended Boyer-Moore theorem prover [SS89] to prove that our queries and transactions are correct. Correctness here would mean that the queries and transactions do what we want them to do. In particular, they preserve the database integrity constraints, and, when desired, they compute only order-independent properties. Thus we can add to Figure 7.6 the class (proved order-independent P) of those queries in SRL, or equivalently in $(\mathrm{FO} + \mathrm{LFP})$ that our theorem-prover has shown to be order-independent.

8. Conclusions

The inference mechanism in [SS89] on finite set terms with variables proves only properties that are true in all models. It can be used to prove that a *set-reduce* expression is independent of order, though it is of necessity incomplete with respect to this problem. (Any algebra meeting the specification is powerful enough to express P problems, and thus the order independence of an arbitrary expression in the language cannot be decided.) Likewise we can prove that some expressions depend on the order. The language of expressions that do not vary with order would have the properties of a specification with an intial algebra, but this language is not recursively enumerable. However, it may be that the set of expressions that their prover can prove are order independent includes all polynomial-time computable, order independent queries. We are investigating this possibility.

Open Problems

1. Problems Related to Ordering:

   (a) Settle Question 7.1. In particular, prove or disprove the conjecture that the subset of SRL that can be proved order-independent using Sheard's Boyer-Moore theorem prover is exactly order-independent P.

   (b) Settle variants of Question 7.1 for smaller complexity classes, e.g. L, NL, NC. Note that for complexity classes NP and above, the question is easily settled because an ordering can simply be existentially quantified and thus no ordering need be provided.

2. Our results show that there is a clear demarcation between SRL which expresses the polynomial-time computable queries and unrestricted SRL which computes all primitive recursive queries. Thus, it is very desirable to improve our characterization of this demarcation line. We would like to be able to say in a very general way, "Yes, *these* sorts of operations and functionalities can all safely be added, without taking us out of P. On the other hand, any of *those* will bring us all the way up to Primitive Recursive complexity."

3. 6.1 shows that to a certain extent the time complexity of an SRL expression can be read off its face. However, we suspect that the complexity bounds we give here can be improved.

4. The classical complexity classes L, NL, P give an interesting basis for comparing the expressibility of query and transaction languages. On the other hand, these are clearly not precisely the complexity classes that are appropriate for studying the true costs of queries and transactions for modern database systems. We are in the process of taking a step in this direction by defining and studying complexity classes more appropriate for database systems. In particular the cost of disk I/O's is given its due place, and incremental complexity is emphasized: we consider the complexity of processing a long sequence of transactions on-line. Much more work is needed in this direction.

References

[AU79] A. Aho, J. Ullman: Universality of data retrieval languages. *Proceedings of Sixth ACM Symposium on POPL*, Jan. 1979, 110-117.

[AK90] S. Abiteboul, P. Kanellakis: Database theory column: Query languages for complex object databases. *SIGACT News 21 No. 3*, Summer 1990, 9-18.

[AK89] S. Abiteboul, P. Kanellakis: Object identity as a query language primitive. *Rapports de Recherche No. 1022, INRIA*, Apr 1989.

[AB88] S. Abiteboul, C. Beeri: On the power of languages for the manipulation of complex objects. *Rapports de Recherche no. 846, INRIA*, May 1988.

[AC89] F. Arafati, S. Cosmadakis: Expressiveness of restricted recursive queries, *Proceedings of 21st ACM STOC*, 1989, 113-126.

[AV88] S. Abiteboul, V. Vianu: Procedural and declarative database update languages. *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on PODS*, 1988, 240-250.

[AV89] S. Abiteboul, V. Vianu: Fixpoint extensions of first-order logic and datalog-like languages. *Proceedings of LICS*, 1989, 2-11.

[AV91] S. Abiteboul, V. Vianu: Generic computation and its complexity. To appear in *32nd IEEE Symposium on FOCS*, 1991.

[BIS88] D. Barrington, N. Immerman, H. Straubing: On uniformity within NC^1. *Journal of Computer Systems and Science 41*, No. 3, 1990, 274-306.

[BM] R.S.Boyer, J.S.Moore:*A Computational Logic*, Academic Press, New York, 1979.

[CFI89] J. Cai, M. Furer, N. Immerman: An optimal lower bound on the number of variables for graph identification. *Proceedings of 30th IEEE Symposium on FOCS*, 1989, 612-617.

[CH80] A. Chandra, D. Harel: Computable queries for relational databases. *Journal of Computer Systems and Science 21*, 1980, 156-178.

[CH82a] A. Chandra, D. Harel: Structure and complexity of relational queries. *Journal of Computer Systems and Science 25*, 1982, 99-128.

[CH82b] A. Chandra, D. Harel: Horn clauses and fixpoint query hierarchy. *Proceedings of 14th ACM STOC*, May 1982, 158-163.

[Ch81] A. Chandra: Programming primitives for database languages. *Proceedings of ACM Symposium on POPL*, 1981, 50-62.

[CSV84] A. Chandra, L. Stockmeyer, U. Vishkin: Constant-depth reducibility, *SIAM Journal of Computing 13*, May 1984, 423-439.

[Co64] A. Cobham: The intrinsic computational difficulty of functions. *Proceedings of the 1964 Congress for Logic, Philosophy and Methodology of Science*, North Holland, 24-30.

[C85] S. Cook: A taxonomy of problems with fast parallel algorithms. *Information and Control 64*, (1985), 2-22.

[CM87] S. Cook, P. McKenzie: Problems complete for deterministic logspace. *Journal of Algorithms 8*, 1987, 385-394.

[CK85] S. Cosmadakis, P. Kanellakis: Parallel evaluation of recursive rule queries. *Proceedings of 5th ACM Symposium on PODS*, 1986, 280-290.

[DW] M. Davis, S. Weyukar: Computability, Complexity and Languages. *Academic Press*, 1983.

[HS89a] R. Hull, J. Su: Untyped sets, invention and computable queries. *Proceedings of 8th ACM Symposium on PODS*, 1989, 347-360.

[HS89b] R. Hull, J. Su: On bulk data type constructors and manipulation primitives - a framework for analyzing expressive power and complexity. *Proceedings of 2nd International Workshop on Database Programming Languages*, June 1989, 396-410.

[HS88] R. Hull, J. Su: On the expressive power of database queries with intermediate types. *Proceedings of 7th ACM Symposium on PODS*, Mar. 1988, 39-51.

[Imm87] N. Immerman: Languages that capture complexity classes. *SIAM Journal on Computing 16 No. 4*, Aug. 1987, 760-778.

[Imm82] N. Immerman: Relational queries computable in polynomial time. *Proceedings of the 14th ACM STOC*, May 1982, 147-152.

[Imm82b] N. Immerman: Upper and lower bounds for first order expressibility. *Journal of Computer and System Sciences 25*, 1982, 76-98.

[Imm88] N. Immermann: Nondeterministic space is closed under complementation. *SIAM J. Comput. 17, No. 5*, (1988), 935-938.

[IL89] N. Immerman, S. Landau: The complexity of iterated multiplication. *Proceedings of 4th Structure in Complexity Theory Conference*, 1989, 104-111.

[IL90] N. Immerman, E. Lander: Describing graphs: a first-order approach to graph canonization. *Complexity Theory Retrospective*, A. Selman, ed., Springer Verlag (1990).

[K88] P. Kanellakis: Elements of relational database theory. *Tech. Report CS-88-09, Dept. Of Computer Science, Brown University*, Apr. 1988.

[OBB89] A. Ohori, P. Buneman, V. Breazu-Tannen: Database programming in Machiavelli - a polymorphic language with static type interference. *Proceedings of the ACM SIGMOD*, June 1989, 46-57.

[Q89] X.Qian: The expressive power of the bounded iteration construct. *Proceedings of the 2nd International Workshop on Database Programming Languages*, 1990.

[SS89] T. Sheard, D. Stemple: Automatic verification of database transaction safety. *ACM transactions on Database Systems 14, No. 3*, Sep. 1989, 322-368.

[Va82] M.Y. Vardi: The complexity of relational query languages. *Proceedings of 14th ACM STOC*, May 1982, 137-146.

[VS90] J. Vitter, E. Shriver: Optimal disk I/O with parallel block transfer. *Proceedings of the 22nd ACM STOC*, May 1990, 159-169.