The Complexity of Reflection

Sushant Patnaik & Jose A. Medina

CMPSCI Technical Report 96-25
April, 1996

Abstract

Reflection in a programming language refers to the ability to generate a program and execute it in the same environment. A well known example is the *eval* construct in LISP. In [VVV93], the authors formally introduced reflection in first-order relational algebra and showed that the complexity of first order relational algebra extended with reflection is exactly PSPACE. We use insights from Descriptive Complexity ([I89]) to show that first-order reductions capture the notion of reflection rather effectively. We prove that the Quantified Boolean Formula problem ([AU79]) is PSPACE complete via quantifier free first order projections. We then use this to get easy and new proofs of the previous results, and answer a question raised in [VVV93] regarding the complexity of higher order reflection. Let $T(1,n) = n^{O(1)}$; $T(i,n) = 2^{T(i-1,n)}$, for $i > 1$. In particular, we show that the complexity of i-th order reflection in first-order calculus exactly equals SPACE$[T(i,n)]$.

1. Introduction

Over the years the notion of reduction has been evolving from using polynomial time bounded transducers to lower complexity transducers such as 1-way logspace or first order. As a consequence many problems that had been known to be complete for a certain complexity class via some kind of reduction such as GAP for Non-deterministic logspace via logspace reduction, or SAT for NP via polylomial time, are now known to be complete for those complexity classes via lower complexity reductions. This is the case of the Quantified Boolean Formulas (QBF) problem as well. Known to be PSPACE complete via a polynomial time reduction [S74], we show that QBF is PSPACE complete via extremely low level reductions, namely, first order projections (in fact, quantifier-free projections). First order projections were introduced first by Immerman [I87] and they are provably weaker than logspace reductions but incomparable to 1-way logspace reductions. Previously, in [S91], Stewart has shown that variants of QBF called CNF_k and DNF_k are complete via first order projections for the classes σ_k^n and π_k^n, respectively. In the first half of the paper, we introduce the descriptive complexity framework and notation for projection reductions in sections 2 and 3 and the QBF completeness proof in section 4.

Next we use the completeness of QBF under such weak reductions to characterize the complexity of a database language construct called *reflection*. The concept of reflection has been around in diverse areas of computer science. In [S*92], [SM*93], the authors define the notion of linguistic reflection, in contrast to behavioral reflection, as the ability of a program to generate code that is integrated into its own execution. The *eval* operation in LISP and SNOBOL-4 and the *popval* function of POP-2 are some examples of run-time reflection. The feature is particularly useful in context of database programming languages, since it provides an elegant way to handle changes to the data, the programs that manipulate the data and the schema. Stemple etal. give numerous other applications and present the practical difficulties of implementing this feature in traditional programming languages. Actual implementation of generation of code and its binding environment and ways of

combining reflection effectively with strong typing and static typing are issues that remain open to debate. This paper investigates the complexity of *run-time linguistic reflection*, in the terminology of [S*92]. Recently in [VVV93], the authors formalize reflection in the relational algebra model, and named it first-order reflection. We shall adopt their approach. Hereafter, unless stated otherwise, by reflection we mean first-order reflection. Note that adding the ability of reflection to a Turing complete language does not enhance its expressiveness. This is no longer true if we consider languages which have smaller expressive complexity[1]. We shall use L both to denote a language and the expressiveness (or, complexity) of the language.

To give an idea of the implementation of "run-time linguistic reflection", we use a pseudo code like notation to describe the procedure EVALUATE, that evaluates an expression in Reflect-L. This is invoked during run-time.

```
COMPILE-EVAL(E:Reflect-L) ==
⋮
proc EVALUATE(E:Reflect-L) ==
⋮
Case
⋮
Occurrence-of-eval(φ,EXIST_φ, ALL_φ, NEG_φ, EQ_φ)): COMPILE-EVAL(GENERATE(
                                                        COMPILE-EVAL(EXIST_φ),
                                                        COMPILE-EVAL(ALL_φ),
                                                        COMPILE-EVAL(NEG_φ),
                                                        COMPILE-EVAL(EQ_φ)));
⋮
end
⋮
```

The procedure GENERATE(L1,L2,L3,L4) takes the encoding of a formula and returns a string in L representing the formula. Note that this function can depend on the input bits and the input size. Note that EXIST_φ, etc. can potentially be Reflect-L expressions, but that takes us into the realm of higher order reflection. We do not allow it in first-order reflection. This restriction is one of syntax and is easily implemented.

Let FO denote the class of problems expressible in fixed size first order logic formulae, or equivalently the safe relational calculus language [[U]] with an ordering on a finite universe. In this model, the universe, U, is $\{1, \ldots, n\}$ and the variables range over this universe.

[1]By (expressive) complexity of a language, we mean the data complexity, i.e. the class of problems that can be expressed in this language.

We assume the presence of an ordering on the universe. Even so, FO is a very weak language. It is well known that we cannot do Parity (Counting number of 1's mod 2) or Transitive Closure in this language. However, in [VVV93], the authors show that the seemingly innocuous operation of *reflection* (defined subsequently), when added to FO, raises its expressiveness all the way to PSPACE. They define a suitable program schema to encode their programs, define variables called program variables and introduce the operation *eval(X)* to evaluate any program variable X. They also add the ability to create unique id numbers by introducing the so-called $\Upsilon_C(r)$ which basically extends a relation r with a new column C containing a distinct new identifier value for each tuple. This gives them the ability to specify an unbounded (i.e. polynomially in the size of the domain, instead of just a constant) length program. Further, they showed with a suitably defined bounded version of reflection, obtained by imposing a syntactic restriction (as shown shortly), the complexity is exactly P (polynomial time). Reflection in a language can be added at a second (and third and so on) order in the following sense: provide constructs that allow a program to generate first order reflective code that can be executed by the program in the same run. They posed as an open question the complexity of adding reflection in higher orders.

Unlike [VVV93], we add *reflection* in the relational calculus framework which is well known to be equivalent to relational algebra. In section 5, we formally define the syntax and semantics of first-order and higher order *reflection* and get simple proofs of their previous results on its expressiveness. Adding reflection to first order logic is analogous to adding a PSPACE generalized quantifier. We generalize the results of [VVV93] using insights from Descriptive Complexity and give a partial answer to their question regarding the complexity of extending first-order calculus by higher order reflection.

2. Coding Quantified Boolean Formulas

Recall the definitions and notation from Chapter ??. We will code all inputs as finite logical structures, i.e., relational databases. A *vocabulary* $\tau = \langle R_1^{a_1} \ldots R_s^{a_s}, c_1, \ldots, c_t \rangle$ is a tuple of input relation and constant symbols. Let STRUCT[τ] denote the set of all finite structures of vocabulary τ. We define a complexity theoretic *problem* to be any subset $S \subseteq$ STRUCT[τ] for some τ.

For example, a n vertex graph would be coded as the structure

$$A = \langle \{0, 1, ..., n-1\}, E^2, k \rangle$$

where E^2 represents the adjacency matrix of the graph. For a given instantiation of E^2 and k, this graph may or may not belong to the problem k-CLIQUE subset of STRUCT[E^2, k] the class of the graphs with a k-size clique as a subgraph.

We code a single tape Turing Machine **M** as:

$$\mathcal{M} = \langle \{0, 1, ..., n-1\}, \mathbf{Q}, \Sigma, \{1, -1, 0\}, \Upsilon_\delta \rangle$$

where $\mathbf{Q}$ is the set of states, $\{1, -1, 0\}$ is the set of movements of the tape head, Σ is the alphabet (assume it is binary) and $\Upsilon_\delta(Q, s_1, P, s_2, N)$ is true if and only if when reading tape symbol s_1 on state Q $\mathbf{M}$ changes to state P, writes symbol s_2, and moves its tape head depending on N's value (-1 for left, 1 for right, 0 for none).

For any vocabulary τ there is a corresponding first-order language $\mathcal{L}(\tau)$ built up from the symbols of τ and the logical relation symbols,$=$ and $\leq$, (which refers to a total ordering on the universe), using logical connectives: $\wedge, \vee, \neg$, variables: $x, y, z, ...$, and quantifiers: $\forall, \exists$.

A quantified Boolean formula is a formula in Boolean logic i. e. the variables and quantifiers are Boolean. In order to code a quantified Boolean formula as a finite structure without loss of generality, we make the following assumptions:

1. For any Boolean variables x, y, we use $x = y$ and $x \neq y$ to abbreviate the formulae $(x \wedge y) \vee (\bar{x} \wedge \bar{y})$ and $(x \vee y) \wedge (\bar{x} \vee \bar{y})$, respectively.
2. The quantifiers are to the left of the formula. The quantifier-free part of the formula is in disjunctive normal form (DNF), where each term of the formula is an equality as given above.[2]
3. We let a single quantifier bind several variables. In the proof of the theorem, quantifiers bind polynomially many variables. We number the leftmost quantifier quantifier 1. We also assume that quantifiers alternate between existential and universal.

We use the following vocabulary to code a quantified boolean formula

$$\langle EQ, NEG, EXIST, ALL \rangle$$

The relations have the following meaning.

$EQ(z, c, e)$ is true when variable z occurs in equality e of clause c.
$NEG(c, e)$ is true when equality e of clause c is negated.
$EXIST(q, z)$ is true when variable z is bound by existential quantifier q.
$ALL(q, z)$ is true when variable z is bound by universal quantifier q.

In general, the argument variables can be vectors.

Example 2.1 We would code the formula

$$\varphi = \exists z_1 \exists z_2 \forall z_3 \forall z_4([(z_1 = z_3) \wedge (z_2 \neq z_3)] \vee [(z_1 \neq z_2) \wedge (z_3 = z_4) \wedge (\neg z_2)])$$

by the following instantiation:

[2] An encoding of the quantified Boolean formula problem using equalities follows the definition in [AHU]

$EQ = \{(1,1,1),(3,1,1),(2,1,2),(3,1,2),(1,2,1),(2,2,1),(3,2,2),(4,2,2),(2,2,3)\}$
$NEG = \{(1,2),(2,1),(2,3),\}$
$EXIST = \{(1,1),(1,2)\}$
$ALL = \{(2,3),(2,4)\}$

3. PSPACE-completeness of QBF

A quantified boolean formula φ is called a *sentence* if it has no free occurrences of any variable. The QBF problem is: given a quantified Boolean sentence, determine if it is true.

Theorem 3.1 *QBF is* PSPACE*-complete via quantifier-free projections.*

Proof: Let M be a Turing Machine with a polynomially bounded single tape with input w. Let us introduce some notation. Q is the set of states of M. Σ is the alphabet with symbols 0 and 1. δ is the transition table of M. Let k be the constant such that M's space is bounded by n^k where n is the size of the input. We represent a configuration of M with polynomially many boolean variables (bits).

$$\langle \mathrm{X}_1^i ... \mathrm{X}_{n^k}^i \ \mathrm{Y}_1^i ... \mathrm{Y}_{n^k}^i \mathrm{Q}_1^i ... \mathrm{Q}_c^i \rangle$$

represents configuration C_i where $\mathrm{X}_1^i ... \mathrm{X}_{n^k}^i$ denote the tape contents, $\mathrm{Y}_1^i ... \mathrm{Y}_{n_k}^i$ denote the head position (it is all zeroes except at the position of the head), and $\mathrm{Q}_1^i ... \mathrm{Q}_c^i$ denote the state of M. Assume M has 2^c states. The transition table δ can be represented by a fixed finite formula $\Upsilon_\delta(\overline{\mathrm{Q}}, z, \overline{\mathrm{Q}'}, z', N)$ that would be true if M when reading symbol z in state $\overline{Q}$, writes symbol z', changes state to $\overline{Q'}$ and executes movement N (-1, 1 or 0). For simplicity we assume that M on each step either writes on the tape (a symbol that is different from the one is one the tape) or moves (left or right).

Given configurations C_1 and C_2, we can check if $C_1 \vdash_M C_2$ with a $O(n^{2k})$ sized formula. Let $C_1 = \langle \overline{\mathrm{X}^1}\ \overline{\mathrm{Y}^1}\ \overline{\mathrm{Q}} \rangle$ and $C_2 = \langle \overline{\mathrm{X}^2}\ \overline{\mathrm{Y}^2}\ \overline{\mathrm{Q}'} \rangle$. The formula is as follows:

$$\gamma = \bigvee_{h=1}^{n^k} \ \bigvee_N (\alpha_l \vee \alpha_r) \bigvee_W \alpha_w$$

where the outermost disjunction locates the position of the head, $\bigvee_N$ is a disjunction over all transitions (a constant number) that move the tape head but do not write onto the tape i. e., $\bigvee_N$ is a disjunction over the set

$$\{(a, \overline{\mathrm{Q}}, a', \overline{\mathrm{Q}'}, N) |\ \Upsilon_\delta(a, \overline{\mathrm{Q}}, a', \overline{\mathrm{Q}'}, N) \ \wedge \ a = a' \ \wedge \ N \text{ is } -1 \text{ or } 1\}$$

α_l checks if C_2 follows from C_1 by moving the head to the left; α_r checks if C_2 follows from C_1 by moving the head to the right; and α_w checks if C_2 follows from C_1 by writing onto the tape on position h. α_l and α_r are similar in form. For example,

$$\alpha_l = (\mathrm{X}_1^1 = \mathrm{X}_1^2 \wedge ... \wedge \mathrm{X}_{n^k}^1 = \mathrm{X}_{n^k}^2) \bigwedge$$
$$(\mathrm{Y}_1^1 = 0 \wedge \mathrm{Y}_1^2 = 0) \wedge ... \wedge (\mathrm{Y}_{h-1}^1 = 0 \wedge \mathrm{Y}_{h-1}^2 = 1) \wedge$$
$$(\mathrm{Y}_h^1 = 1 \wedge \mathrm{Y}_h^2 = 0) \wedge ... \wedge (\mathrm{Y}_{n^k}^1 = 0 \wedge \mathrm{Y}_{n^k}^2 = 0)$$
$$\langle \mathrm{Q}_1^1 \mathrm{Q}_2^1 ... \mathrm{Q}_{n^k}^1 \rangle = \overline{\mathrm{Q}} \ \wedge \ \mathrm{X}_h^1 = a$$
$$\langle \mathrm{Q}_1^2 \mathrm{Q}_2^2 ... \mathrm{Q}_{n^k}^2 \rangle = \overline{\mathrm{Q}'} \ \wedge \ \mathrm{X}_h^2 = a$$

Basically $\mathrm{X}_h^1 = 1$ in C_1 changes to $\mathrm{X}_h^2 = 0$ in C_2 and $\mathrm{X}_{h-1}^1 = 0$ changes to $\mathrm{X}_h^2 = 1$.

$\bigvee_W$ is a disjunction over all transitions (also a constant number) that write onto the tape but do not move the tape head i. e., it is a disjunction over the set

$$\{(a, \overline{\mathrm{Q}}, a', \overline{\mathrm{Q}'}, N) | \ \Upsilon_\delta(a, \overline{\mathrm{Q}}, a', \overline{\mathrm{Q}'}, N) \ \wedge \ a \neq a' \ \wedge \ N \text{ is } 0\}$$

and α_w is as follows:

$$\alpha_w = (\mathrm{X}_1^1 = \mathrm{X}_1^2) \wedge ... \wedge (\mathrm{X}_h^1 \neq \mathrm{X}_h^2) \wedge (\mathrm{X}_{n^k}^1 = \mathrm{X}_{n^k}^2) \bigwedge$$
$$(\mathrm{Y}_1^1 = 0 \wedge \mathrm{Y}_1^2 = 0) \wedge ... \wedge (\mathrm{Y}_h^1 = 1 \wedge \mathrm{Y}_h^2 = 1) \wedge ... (\mathrm{Y}_{n^k}^1 = 0 \wedge \mathrm{Y}_{n^k}^2 = 0)$$
$$\langle \mathrm{Q}_1^1 \mathrm{Q}_2^1 ... \mathrm{Q}_{n^k}^1 \rangle = \overline{\mathrm{Q}} \wedge \mathrm{X}_h^1 = a$$
$$\langle \mathrm{Q}_1^2 \mathrm{Q}_2^2 ... \mathrm{Q}_{n^k}^2 \rangle = \overline{\mathrm{Q}'} \wedge \mathrm{X}_h^2 = a'$$

The symbol a at position h in C_1 changes to a' in C_2.

Let C_{init} denote the initial configuration of M and C_{accept} its final accepting configuration. Let $T(C^i, C^j, t)$ be the quantified Boolean formula that has value true when configuration C^j is reachable from configuration C^i in 2^t steps of M. Thus, we can check a valid accepting computation of M with input w with the formula:

$$\exists \overline{\mathrm{Z}_1^0} \exists \overline{\mathrm{Z}_2^0} (\overline{\mathrm{Z}_1^0} = C_{init} \wedge \overline{\mathrm{Z}_1^0} = C_{accept} \wedge T(\overline{\mathrm{Z}_1^0}, \overline{\mathrm{Z}_2^0}, n^k))$$

Let Φ denote the above formula. T is denested in this formula until the third argument is 0. Next, in order to avoid getting an exponential size formula, variables are renamed. Thus we have:

$$T(\overline{\mathrm{Z}_1^0}, \overline{\mathrm{Z}_2^0}, n^k) = \exists \overline{\mathrm{Z}_0^1} \forall \overline{\mathrm{Z}_1^1} \forall \overline{\mathrm{Z}_2^1}$$
$$\left[(\overline{\mathrm{Z}_1^1} = \overline{\mathrm{Z}_1^0} \wedge \overline{\mathrm{Z}_2^1} = \overline{\mathrm{Z}_0^1}) \vee (\overline{\mathrm{Z}_1^1} = \overline{\mathrm{Z}_0^1} \wedge \overline{\mathrm{Z}_2^1} = \overline{\mathrm{Z}_2^0}) \Rightarrow T(\overline{\mathrm{Z}_1^1}, \overline{\mathrm{Z}_2^1}, n^k - 1) \right]$$

If we write the quantifier-free part of the formula in disjunctive normal form, we have:

$$T(\overline{\mathrm{Z}_1^0}, \overline{\mathrm{Z}_2^0}, n^k) = \exists \overline{\mathrm{Z}_0^1} \forall \overline{\mathrm{Z}_1^1} \forall \overline{\mathrm{Z}_2^1} [\ (\overline{\mathrm{Z}_1^1} \neq \overline{\mathrm{Z}_1^0} \wedge \overline{\mathrm{Z}_1^1} \neq \overline{\mathrm{Z}_0^1}) \vee (\overline{\mathrm{Z}_1^1} \neq \overline{\mathrm{Z}_1^0} \wedge \overline{\mathrm{Z}_2^1} \neq \overline{\mathrm{Z}_2^0}) \vee$$
$$(\overline{\mathrm{Z}_2^1} \neq \overline{\mathrm{Z}_0^1} \wedge \overline{\mathrm{Z}_1^1} \neq \overline{\mathrm{Z}_0^1}) \vee (\overline{\mathrm{Z}_2^1} \neq \overline{\mathrm{Z}_0^1} \wedge \overline{\mathrm{Z}_2^1} \neq \overline{\mathrm{Z}_2^0}) \vee$$
$$T(\overline{\mathrm{Z}_1^1}, \overline{\mathrm{Z}_2^1}, n^k - 1)]$$

For example, we show $T(\overline{Z_1^1}, \overline{Z_2^1}, n^k - 1)$.

$$T(\overline{Z_1^0}, \overline{Z_2^0}, n^k - 1) = \exists \overline{Z_0^2} \forall \overline{Z_1^2} \forall \overline{Z_2^2} [\ (\overline{Z_1^2} \neq \overline{Z_1^1} \wedge \overline{Z_1^2} \neq \overline{Z_0^2}) \vee (\overline{Z_1^2} \neq \overline{Z_1^1} \wedge \overline{Z_2^2} \neq \overline{Z_2^1}) \vee (\overline{Z_2^2} \neq \overline{Z_2^2} \wedge \overline{Z_1^2} \neq \overline{Z_0^2}) \vee (\overline{Z_2^2} \neq \overline{Z_0^2} \wedge \overline{Z_2^2} \neq \overline{Z_2^1}) \vee T(\overline{Z_1^2}, \overline{Z_2^2}, n^k - 2)]$$

And so on until we reach $T(\overline{Z_1^{n^k-1}}, \overline{Z_2^{n^k-1}}, 0)$.

After denesting Φ we have:

$$\exists \overline{Z_1^0} \exists \overline{Z_2^0} \left(\overline{Z_1^0} = C_{init} \wedge \overline{Z_1^0} = C_{accept} \wedge \left[Q_1 \cdots Q_{2n^k} \left(\overline{C_1} \vee \cdots \vee \overline{C_{4n^k}} \right) T(\overline{Z_1^{n^k-1}}, \overline{Z_2^{n^k-1}}, 0) \right] \right)$$

Finally, Φ is massaged into a DNF formula as follows:

$$\exists \overline{Z_1^0} \exists \overline{Z_2^0} Q_1 \cdots Q_{2n^k} \left[\overline{Z_1^0} = C_{init} \wedge \overline{Z_1^0} = C_{accept} \wedge \overline{C_1} \wedge T(\overline{Z_1^{n^k-1}}, \overline{Z_2^{n^k-1}}, 0) \right] \vee \cdots \vee \left[\overline{Z_1^0} = C_{init} \wedge \overline{Z_1^0} = C_{accept} \wedge \overline{C_{4n^k}} \wedge T(\overline{Z_1^{n^k-1}}, \overline{Z_2^{n^k-1}}, 0) \right]$$

The size of the formula Φ is $O(n^{2k})$, and Φ is **true** if and only if M accepts w. We now show that this reduction is a quantifier-free projection from $L(\mathrm{M})$ to QBF.

Next we show a first order projection for each one of $\varphi_=, \varphi_\neg, \varphi_\exists, \varphi_\forall$, corresponding respectively to $EQ, NEG, EXIST, ALL$, the relations of the structure. We need an encoding to index the variables and quantifiers of Φ. Each of the symbols $\overline{Z_i^j}$ in the formula represents $2 \cdot n^k + c$ boolean variables. Each of these variables is indexed by $\langle j_1 \cdots j_k, i, t_1 \cdots t_k, p \rangle$ where $j_1 \cdots j_k$ and $t_1 \cdots t_k$ represent numbers between 0 and n_k. $\bar{j}$ refers to the nesting level where the variable is bound (this $\bar{j}$ refers to the j in $\overline{Z_i^j}$, this j goes from 0 to $n^k - 1$). i refers to the subscript of $\overline{Z_i^j}$ $(0 \leq i \leq 2)$. $\bar{t}$ serves as an index whose meaning depends on the value of p. If $p = 0$, then $\bar{t}$ is an index for the variables representing the contents of the tape; if $p = 1$, $\bar{t}$ is an index for the variables representing the position of the head; and if $p = 2$, it is an index for the variables representing the state of M.

Clauses of Φ are indexed by $\langle c_1 \cdots c_k, (a, Q, a', Q', N), d, s \rangle$ in a similar fashion. When $s = 0$, $c_1 \cdots c_k$ names the clause in the quantifier-free part of Φ that appears when denesting T at step $c_1 \cdots c_k$, and $1 \leq d \leq 4$ covers all the four clauses that are obtained at each denesting step. If $s = 1$, $c_1 \cdots c_k$ indexes the possible head position when we check if a configuration is reachable from another one in one step. Considering the formula, γ above, $c_1 \cdots c_k$ indexes the leftmost disjunction; (a, Q, a', Q', N) indexes the disjunction within $\bigvee_N$ or $\bigvee_W$; and $d = 1, 2, 3$ indexes the three cases corresponding to α_l, α_r or α_w, respectively.

Recall that each clause comprises of equalities. An equality is indexed by $\langle \bar{c}, t_1 \cdots t_k, p, J, flag, a, Q, a', Q' \rangle$. $\bar{c}$ indexes the clause in which the equality appears. $\langle t_1 \cdots t_k, p \rangle$ gives the position within $\bar{c}$; $J = 0$ indicates it is an equality in the C_{init} part (look at DNF of Φ above); $J = 1$ indicates it is an equality in the C_{accept} part; $J = 2$ indicates it is an equality in the first conjunct of C_l; $J = 3$ indicates it is an equality in the second conjunct of C_l; $J = 4$ indicates it is an equality in the $T(\overline{Z_1^{n^k-1}}, \overline{Z_2^{n^k-1}}, 0)$ part.

For indexing quantifiers, we use the same tuple that we use for indexing clauses, since exactly one existential quantifier and one universal quantifier appear at each denesting step.

At this point we are ready to present the quantifier-free projection from L(M) to QBF.

I. $EXIST\ (\langle j_1 \cdots j_k, i, t_1 \cdots t_k, p \rangle, \langle c_1 \cdots c_k, (a, Q, a', Q', N), d, s \rangle) \equiv \varphi_\exists$

II. $ALL\ (\langle j_1 \cdots j_k, i, t_1 \cdots t_k, p \rangle, \langle c_1 \cdots c_k, (a, Q, a', Q', N), d, s \rangle) \equiv \varphi_\forall$

III. $NEG\ (\langle c_1 \cdots c_k, (a, Q, a', Q', N), d, s \rangle,$
$\langle \langle c'_1 \cdots c'_k, (a_0, Q_0, a'_0, Q'_0, N_0), d', s' \rangle, t_1 \cdots t_k, p, J, flag, (a_1, Q_1, a'_1, Q'_1) \rangle \rangle$
$\equiv \varphi_\neg$

III. $EQ\ (\langle j_1 \cdots j_k, i, t_1 \cdots t_k, p \rangle,$
$\langle c_1 \cdots c_k, (a, Q, a', Q', N), d, s \rangle,$
$\langle \langle c'_1 \cdots c'_k, (a_0, Q_0, a'_0, Q'_0, N_0), d', s' \rangle, t'_1 \cdots t'_k, p, J, flag, (a_1, Q_1, a'_1, Q'_1) \rangle \rangle$
$\equiv \varphi_=$

Where $\varphi_\forall$, $\varphi_\exists$, $\varphi_\neg$, $\varphi_=$ are given by:

$\varphi_\exists = (\bar{j} = \bar{c} \neq \bar{0} \wedge i = 0 \wedge s = 0) \vee (\bar{j} = \bar{q} = \bar{0} \wedge (i = 1 \vee i = 2))$

$\varphi_\forall = (\bar{j} = \bar{c} \wedge p = 0 \wedge i \neq 0)$

$$\varphi_\neg = (\bar{c} = \bar{c'} \wedge (s = 0) \wedge (1 \leq d \leq 4) \wedge (J = 2 \vee J = 3)$$
$$\bigvee$$
$$(J = 4) \wedge (d = 3) \wedge \langle c_1 \cdots c_k, (a, Q, a', Q', N), d, s \rangle =$$
$$\langle c'_1 \cdots c'_k, (a_0, Q_0, a'_0, Q'_0, N_0), d', s' \rangle \wedge (s = 1) \wedge$$
$$c_1 \cdots c_k = t_1 \cdots t_k \wedge (p = 0) \wedge (a \neq a')$$

$$\varphi_= = (0 \leq flag \leq 2) \wedge (0 \leq a, a' \leq 1) \wedge (1 \leq Q \leq 2^c) \wedge$$
$$(0 \leq p \leq 2) \wedge (1 \leq d \leq 4) \wedge (0 \leq s \leq 1) \wedge (0 \leq i \leq 4) \wedge (1 \leq J \leq 4) \wedge$$
$$(\alpha_{j_0} \vee \alpha_{j_1} \vee \alpha_{j_2} \vee \alpha_{j_3} \vee \alpha_{j_4})$$

and α_{j_0}, α_{j_1}, α_{j_2}, α_{j_3} and α_{j_4} are given by

$$\begin{aligned}\alpha_{j_0} = (J=0) \wedge j_1 \cdots j_k = \overline{0} \wedge t_1 \cdots t_k p = t'_1 \cdots t'_k p' \wedge \\ [(i=1) \vee \\ (i=3 \wedge p=2) \vee \\ (p=0 \wedge i=3 \wedge t_2 \cdots t_k = \overline{0} \wedge \neg w(t_1)) \vee \\ (p=0 \wedge i=4 \wedge t_2 \cdots t_k = \overline{0} \wedge w(t_1)) \vee \\ (p=1 \wedge i=4 \wedge t_1 \cdots t_k = \overline{0}) \vee \\ (p=1 \wedge i=3 \wedge t_1 \cdots t_k \neq \overline{0}) \vee \\ (p=0 \wedge i=3 \wedge t_1 \cdots t_k \neq \overline{0})]\end{aligned}$$

$$\begin{aligned}\alpha_{j_1} = (J=1) \wedge j_1 \cdots j_k = \overline{0} \wedge t_1 \cdots t_k p = t'_1 \cdots t'_k p \wedge \\ [(i=2) \vee \\ (i=3 \wedge p=0) \vee \\ (p=1 \wedge i=4 \wedge t_1 \cdots t_k = \overline{0}) \vee \\ (p=1 \wedge i=3 \wedge t_1 \cdots t_k \neq \overline{0}) \vee \\ (p=2 \wedge i=4)]\end{aligned}$$

$$\begin{aligned}\alpha_{j_2} = (J=2) \wedge (s=0) \wedge t_1 \cdots t_k p = t'_1 \cdots t'_k p \wedge \\ \{c_1 \cdots c_k = c'_1 \cdots c'_k = j_1 \cdots j_k \wedge \\ [(d=1) \wedge (i=0 \vee i=1) \vee \\ (d=2) \wedge (i=1 \vee i=2) \vee \\ (d=3) \wedge (i=0 \vee i=1 \vee i=2) \vee \\ (d=4) \wedge (i=0 \vee i=2)] \\ \vee \\ [c_1 \cdots c_k - 1 = c'_1 \cdots c'_k = j_1 \cdots j_k \wedge \\ [(d=1) \wedge (i=1) \vee \\ (d=2) \wedge (i=1 \vee i=2) \vee \\ (d=4) \wedge (i=2)]\}\end{aligned}$$

$$\begin{aligned}\alpha_{j_3} = (J=3) \wedge (s=0) \wedge t_1 \cdots t_k p = t'_1 \cdots t'_k p \wedge \\ \{c_1 \cdots c_k = c'_1 \cdots c'_k = j_1 \cdots j_k \wedge \\ [(d=1) \wedge (i=0 \vee i=1) \vee \\ (d=2) \wedge (i=1 \vee i=2) \vee \\ (d=3) \wedge (i=0 \vee i=1 \vee i=2) \vee \\ (d=4) \wedge (i=0 \vee i=2)] \\ \vee\end{aligned}$$

$$[c_1 \cdots c_k - 1 = c'_1 \cdots c'_k = j_1 \cdots j_k \wedge$$
$$(d = 2) \wedge (i = 1 \vee i = 2) \vee$$
$$(d = 4) \wedge (i = 2)]\}$$

$$\alpha_{j_4} = (J = 4) \wedge (s = 1) \wedge a = a_0 \wedge Q = Q_0 \wedge a' = a'_0 \wedge Q' = Q'_0 \; wedge N_0 = N'_0$$
$$j_1 \cdots j_k = \overline{n} \wedge c_1 \cdots c_k = c'_1 \cdots c'_k \wedge t_1 \cdots t_k = t'_1 \cdots t'_k \wedge \Upsilon_\delta(a, Q, a', Q', 0) \wedge N_0 = 0$$
$$\{p = 0 \wedge (i = 1 \vee i = 2)$$
$$\vee$$
$$p = 1 \wedge flag = 0 \wedge$$
$$[i = 1 \vee (i = 3 \wedge t_1 \cdots t_k \neq t'_1 \cdots t'_k) \vee (i = 4 \wedge t_1 \cdots t_k = t'_1 \cdots t'_k)]$$
$$\vee$$
$$p = 1 \wedge flag = 1 \wedge d = 2 \wedge \Upsilon_\delta(a, Q, a', Q') \wedge$$
$$[i = 2 \vee (i = 4 \wedge t'_1 \cdots t'_k = c_1 \cdots c_k - 1) \vee (i = 3 \wedge t'_1 \cdots t'_k = c_1 \cdots c_k - 1)]$$
$$\vee$$
$$p = 1 \wedge flag = 1 \wedge d = 2 \wedge \Upsilon_\delta(a, Q, a', Q', -1) \wedge N_0 = -1$$
$$[i = 2 \vee (i = 4 \wedge t'_1 \cdots t'_k = c_1 \cdots c_k + 1) \vee (i = 3 \wedge t'_1 \cdots t'_k = c_1 \cdots c_k + 1)]$$
$$\vee$$
$$p = 2 \wedge [flag = 0 \wedge (i = 1 \vee (i = 3 \wedge \neg BIT(t'_1, Q)) \vee (i = 4 \wedge BIT(t'_1, Q))) \vee$$
$$flag = 1 \wedge (i = 2 \vee (i = 3 \wedge \neg BIT(t'_1, Q)) \vee (i = 4 \wedge BIT(t'_1, Q)))]$$
$$\vee$$
$$p = 1 \wedge flag = 1 \wedge d = 3 \wedge \Upsilon_\delta(a, Q, a', Q', 1) \wedge N_0 = 1$$
$$[i = 2 \vee (i = 4 \wedge t'_1 \cdots t'_k = c_1 \cdots c_k) \vee (i = 3 \wedge t'_1 \cdots t'_k \neq c_1 \cdots c_k)]$$
$$\vee$$
$$p = 0 \wedge flag = 2 \wedge t'_1 \cdots t'_k = c_1 \cdots c_k \wedge$$
$$[i = 1 \vee (i = 3 \wedge a = 0) \vee (i = 4 \wedge a = 1)]\} \wedge$$
$$[(a = a' \wedge d \neq 3) \vee (a \neq a' \wedge d = 3)]$$

The proof is now complete by noting that the quantifier-free formulae when massaged to disjunctive normal forms are basically projections. ∎

4. Definitions and Reflection Constructs

Formally, we incorporate reflection in the language of first-order calculus by allowing an operation

$$\text{eval}(\varphi, \text{EXIST}_\varphi(q, z), \text{ALL}_\varphi(c, e), \text{NEG}_\varphi(q, z), \text{EQ}_\varphi(z, c, e))$$

where φ is a variable that represents a quantified Boolean sentence (with no free variables, for technical convenience) and the first-order logic expressions, $\text{EXIST}_\varphi, \text{ALL}_\varphi, \text{NEG}_\varphi$, and EQ_φ, encode the formula, φ (analogous to the program schema of [VVV93]), as shown

below. In the sequel, for notational convenience, we shall frequently drop the variables from our expressions.

We use the the unary domain relations VARIABLE, CLAUSE, EQUALITY, QUANT, FORMULA-VARIABLE whose meanings stem from their names and constants 0 and 1, if necessary. For all $z \in$ VARIABLE, $c \in$ CLAUSE, $e \in$ EQUALITY, $q \in$ QUANT and $\varphi \in$ FORMULA-VARIABLE.

$EQ_\varphi(z, c, e)$ is true if variable z occurs in equality e of clause c in formula φ.
$NEG_\varphi(c, e)$ is true if equality e of clause c is negated.
$EXIST_\varphi(q, z)$ is true if variable z is bound by existential quantifier q.
$ALL_\varphi(q, z)$ is true if variable z is bound by universal quantifier q.

In general, the variables in these relations can be fixed length vectors. The subscript will be dropped when the formula variable is clear from the context.

We shall assume for convenience that the formula φ specified in *eval* is in prenex form. We further assume that the quantifier-free part of the formula is in disjunctive normal form (or, conjunctive normal form).

The semantics of

$$\text{eval}(\varphi, \text{EXIST}_\varphi(q, z), \text{ALL}_\varphi(c, e), \text{NEG}_\varphi(q, z), \text{EQ}_\varphi(z, c, e))$$

are simple : by cycling through all possible assignments to the variables q, z, e, c, φ is generated from its encoding relations and evaluated returning a Boolean value: True or False.

In particular, *eval* returns a valid result (1 or 0) only for syntactically correct φ and is undefined (*error*) otherwise. We shall consider only syntactically correct formulae. Note that, unlike [VVV93], we do not need an analogue to the Γ_C operator, since we have an ordering on the domain and we can specify a polynomial length, say n^k, by simply using k first-order variables (note that, by definition, any first-order variable takes values in $\{0, \ldots, n-1\}$). Most of the time in proofs, we shall be using formulae in quantified Boolean logic, a subclass of first-order logic i. e. the variables in the formulae are Boolean valued and the quantifiers are Boolean.

Definition 4.1 We define Reflect-FO as the language of first-order logic equipped with the *eval* operator. In general, for any language L, let L also denote the class of decision problems expressible in L (i. e. the complexity of the language itself), and let *Reflect-L* denote the class of decision problems expressible in language L extended with the reflection construct, *eval*.

Definition 4.2 Consider the following syntactic restriction on any formula φ that can be used in an *eval* operation: φ is allowed to use a finite fixed number (constant independent

of n, the size of the universe) of variables in its body. Let *BoundReflect-L* denote the class of decision problems expressible in language L extended with this restricted *eval* operation.

Unless stated otherwise, by reflection we mean first-order reflection. Using the notion of FO reductions and Immerman's characterizations in descriptive complexity theory ([I86, I87, I89]), we can exactly determine the expressive complexity of a large class of languages when extended with the *reflection* operators.

Theorem 4.3 *Let* L *be any language such that* FO$\subseteq$ L. *Then,* Reflect-L *contains* PSPACE.

The semantics of *reflection* impart an inordinate degree of expressiveness to a language. This is not surprising, since reflection basically extends first-order logic with a PSPACE-complete generalized quantifier.

Corollary 4.4 *Let* L *be any language such that* FO$\subseteq$ L $\subseteq$ P. *Then,* Reflect-L = PSPACE.

Using Immerman's characterization of P in [I89], we can show that

Theorem 4.5 *Let* L *be any language such that* FO$\subseteq$ L. *Then,* BoundReflect-L *contains* *P*.

Corollary 4.6 *Let* L *be any language such that* FO$\subseteq$ L $\subseteq$ *P*. *Then,* BoundReflect-L = *P*.

We note that bounded reflection does not add to the expressiveness of a PSPACE complete language. Abusing notation slightly, and keeping in mind that we use L both to denote the language and the class of decision problems expressible in it, we conclude that

Proposition 4.7 BoundReflect-PSPACE = PSPACE.

In [VVV93], the authors previously proved corollaries 4.4 and 4.6 for FO, albeit differently, and they posed a question about the expressiveness of second and higher order reflection. Incorporating higher order reflection in this framework with appropriate semantics and syntax seems to be a problem. One possible way of achieving it is as follows. Second-order reflection is implemented by allowing the formula φ itself, in any *eval* operation, to be encoded by Reflect-FO expressions. Reflection of any higher order can be defined similarly.

Definition 4.8 Let *second-order reflection* denote the ability to evaluate formulae in first order logic with Equality that are themselves *specified* by Reflect-FO formulae. The syntax of the second order reflection operator is

$$eval(\varphi, \text{EXIST}_\varphi, \text{ALL}_\varphi, \text{EQ}_\varphi, \text{NEG}_\varphi)$$

where the formulae for EXIST_φ, ALL_φ, EQ_φ and NEG_φ are Reflect-FO expressions.

The semantics are similar to Reflect-FO: the Reflect-FO expressions for each of the relations, EXIST, ALL, NEG, EQ, that encode φ are evaluated and used to generate the quantified Boolean sentence, φ, and then φ is evaluated to give True or False.

We denote this language as Reflect-Reflect-FO, or, Reflect^2-FO. Iterating this definition in an obvious way, we get Reflect^k-FO, for any positive integer k.

Example 4.9 Suppose, for some quantified Boolean sentence, ψ, each of its encoding relations, R $\in$ { EXIST, NEG, ALL, EQ } can be specified by a Reflect-FO formula, namely,

$$eval(\varphi, \text{EXIST}_{\varphi_R}(q,z), \text{ALL}_{\varphi_R}(c,e), \text{NEG}_{\varphi_R}(q,z), \text{EQ}_{\varphi_R}(z,c,e))$$

where the relations, EXIST_{φ_R}, ALL_{φ_R}, EQ_{φ_R}, NEG_{φ_R}), are first-order formulae over a universe, $\{0,\ldots,n-1\}$. Then, the following Reflect^2-FO expression, (we drop the variables due to notational convenience)

$eval(\psi, eval(\varphi_1, \text{EXIST}_{\varphi_1}, \text{ALL}_{\varphi_1}, \text{EQ}_{\varphi_1}, \text{NEG}_{\varphi_1}), eval(\varphi_2, \text{EXIST}_{\varphi_2}, \text{ALL}_{\varphi_2}, \text{EQ}_{\varphi_2}, \text{NEG}_{\varphi_2}),$
$eval(\varphi_3, \text{EXIST}_{\varphi_3}, \text{ALL}_{\varphi_3}, \text{EQ}_{\varphi_3}, \text{NEG}_{\varphi_3}), eval(\varphi_4, \text{EXIST}_{\varphi_4}, \text{ALL}_{\varphi_4}, \text{EQ}_{\varphi_4}, \text{NEG}_{\varphi_4}))$

evaluates the formula ψ and returns True or False. Note that the size of the Reflect^2-FO formula is constant, but the size of ψ used in the *eval* operator might be $2^{n^{O(1)}}$, since a Reflect-FO formula, can specify a $2^{n^{O(1)}}$ length input in PSPACE.

Let EXPSPACE denote the class of decision problems in SPACE[2^{n^k}]. Using the well known fact that evaluating exponential sized quantified Boolean sentences is complete for EXPSPACE under PSPACE reductions, we can show that

Theorem 4.10 EXPSPACE = Reflect^2-FO.

Let

$$T(i,n) = 2^{\cdots^{2^{n^{O(1)}}}}$$

where the height of the tower of 2's is i. In particular, $T(1,n) = 2^{n^k}$. Barring technical details, essentially the same argument establishes that

Theorem 4.11 *For any $i > 0$, Reflect^i-FO= SPACE[$T(i,n)$].*

5. Proofs

Proof: (of Theorem 4.3)

(PSPACE in Reflect-L): We use the notion of FO reductions to show that any PSPACE problem can be expressed in Reflect-FO. We start with an easy technical observation.

Lemma 5.1 Let A,B be structures with relations $R_1,\ldots,R_a$, and $S_1,\ldots,S_b$ respectively, where a,b are constants. Then, A $\leq_{FO}$ B implies that for every j, there exists a FO formula ψ_{S_j} with atoms drawn from the relations R_i such that $\psi_{S_j}(\bar{x})$ iff $S_j(\bar{x})$.

By Theorem 3.1, Quantified Boolean Formula is complete for PSPACE under *quantifier-free projections*. Thus, given any PSPACE Turing machine M and its input w, we can construct an instance of a QBF, say, Q i. e. we have the formulae in first-order logic defined over w and Υ_{delta}, the finitely specified transition relation of M: ψ_{EXIST}, ψ_{ALL}, ψ_{NEG}, ψ_{EQ}, that give us the encoding: EXIST_Q, ALL_Q, NEG_Q and EQ_Q. We can then express the PSPACE computation as the following expression in Reflect-FO :

$$E = eval(Q, \psi_{EXIST_Q}(\bar{z},\bar{q}), \psi_{ALL_Q}(\bar{z},\bar{q}), \psi_{NEG_Q}(\bar{e},\bar{c}), \psi_{EQ_Q}(\bar{z},\bar{c},\bar{e}))$$

Clearly, E is true iff Q is true iff M accepts w. ∎

Actually the proof shows that adding an *eval*-like operation to any language L that can express polynomial length quantifier free Boolean sentences suffices to express PSPACE.

Proof: (Proof of corollary 4.4)

(Reflect-L in PSPACE): Since L $\subseteq$ P, and by definition, *eval* only allows as its argument a polynomial (in n) length expression φ, it can be simulated in PSPACE. Further, at most polynomially many different φ can be expressed in this language and composed at most polynomial number of times. This does not cause any problem since we are concerned only with (0/1) decision problems. ∎

Proof: (Sketch of theorem 4.5:BoundReflect-L contains P)

We use Immerman's characterization of P.

P

Fact 5.2 ([I89]) $= \text{FO}[n^{O(1)}]$.

$\text{FO}[n^k]$ essentially means a fixed first-order alternating universal and existential quantifier block iterated $n^O(1)$ times, where n, as usual, denotes the size of the universe. Refer [I89] for the definitions.

Given any problem P in P, the above theorem gives a constant sized FO formula block, call it QB_P. QB_P has a very uniform structure depending on the polytime Turing Machine's program and not on the input. All we have to do to achieve the iteration in Reflect-L is to construct a FO formula Q, that consists of $n^{O(1)}$ copies of QB_P concatenated together in sequence, and then evaluate it using $eval(Q)$. The formulae, EXIST_Q, ALL_Q, NEG_Q and EQ_Q, encoding the formula Q are readily described by quantifier-free projections. The encoding is straightforward but tedious and we omit it in the interests of brevity. ∎

Proof: (of corollary 4.6: BoundReflect-L is in P)

Since L $\subseteq$ P and *eval* can only evaluate expressions with a fixed number of variables, we can have at most polynomially many formulae with fixed number of variables composed polynomial number of times. This can be simulated easily in P. ∎

Proof: (of proposition 4.7)

It suffices to show that PSPACE contains BoundReflect-PSPACE. In PSPACE one can iterate at most 2^{n^c} for some constant c. So at most a 2^{n^c} size quantified Boolean formula, ψ, can be encoded and evaluated usin *eval*. But, any 2^{n^c} sized expression with a fixed number of variables, say k, can be evaluated in PSPACE, since k Boolean variables have only 2^k possible assignments and we use a polynomial sized counter to walk through the expression. ∎

Proof: (of theorem 4.10)

The proof is fairly straightforward. Any Reflect^2-FO expression can be evaluated in EXPSPACE, since, as indicated in Example 4.9, the size of any quantified Boolean sentence evaluated in an *eval* operation is at most $2^{n^{O(1)}}$, where n is the size of the domain.

Given any EXPSPACE Turing machine, M, and an input string w ($|w| = n$), we can construct in PSPACE a 2^{n^k} sized qbf, Q, that describes the computation of M on w (similar to the construction in FO of a polynomial sized qbf for PSPACE as in Theorem 3.1), where k is some constant. Since this construction can be done in PSPACE, there exist $n^{k'}$ sized quantified Boolean formulae, $\mathrm{R}_1, \ldots, \mathrm{R}_4$, that compute, respectively, the encoding relations, EXIST_Q, ALL_Q, NEG_Q, EQ_Q, of formula, Q, for k' a constant.

Hence, by Theorem 4.3, we have Reflect-FO expressions

$$\begin{aligned} eval(R_1, S_{1,1}(\bar{x}), \ldots, S_{1,4}(\bar{x})) &\equiv \mathrm{EXIST}_Q \\ eval(R_2, S_{2,1}(\bar{x}), \ldots, S_{2,4}(\bar{x})) &\equiv \mathrm{ALL}_Q \\ eval(R_3, S_{3,1}(\bar{x}), \ldots, S_{3,4}(\bar{x})) &\equiv \mathrm{NEG}_Q \\ eval(R_4, S_{4,1}(\bar{x}), \ldots, S_{4,4}(\bar{x})) &\equiv \mathrm{EQ}_Q \end{aligned}$$

where $\mathrm{S}_{i,1}$, $\mathrm{S}_{i,2}$, $\mathrm{S}_{i,3}$, $\mathrm{S}_{i,4}$ are quantifier-free projections of maximum arity k'' (all can be assumed to have the same arity), where k'' is a fixed constant. Note that the syntax and semantics of Reflect^2-FO allows the implicit representation of the variables, $\bar{c}, \bar{z}, \bar{e}, \bar{q}$ for

EXIST_Q, ALL_Q, NEG_Q, EQ_Q. (Each of these variables is of length at most polynomial in n.)

Then, the following Reflect^2-FO expression,

$$eval(Q, eval(R_1, S_{1,1}(\bar{x}), \ldots, S_{1,4}(\bar{x})), \ldots, eval(R_4, S_{4,1}(\bar{x}), \ldots, S_{4,4}(\bar{x})))$$

is true iff M accepts w, where $\bar{x}$ is a vector of first-order variables of length k''. ∎

Proof: (Proof of theorem 4.11)

This follows from the observation that the above construction can be repeated. We omit the straightforward details. ∎

6. Conclusion

An open question is to determine what is the expressiveness of adding a suitable version of *reflection* operator (and higher order, perhaps) to FO +(*while*), or equivalently, adding *reflection* to any language whose complexity is exactly PSPACE. In fact, it is not clear how to add this construct, *eval*(φ), to an arbitrary language, say L, with a suitable operational syntactic and semantic definition. One way is to allow the φ to be specified only by first-order logic formulae as in the results above, unlike the general method where φ can be specified by any program in L.

These results would be much improved if they categorized languages in terms of Database Complexity Classes rather than traditional complexity classes. Dynamic Computational complexity measures are more relevant for database (or disk access bound) computation.

7. Acknowledgements

We thank Kousha Etessami for sharing his observations about the QBF proof and Dave Stemple for pointing out the problem of expressiveness of reflection and telling us about the paper by [VVV93]. Finally, we are grateful to Neil Immerman for his constant encouragement and support.

References

[AHU] A. Aho, J. Hopcroft and J. Ullman. *The design and analysis of algorithms*, McGraw Hill, 1979.

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

[I86] N. Immerman, "Relational Queries Computable in Polynomial Time," *Information and Control,* **68** (1986), 86-104. A preliminary version appeared in *14th ACM STOC Symp.* (1982), 147-152.

[I87] N. Immerman, "Languages That Capture Complexity Classes," *SIAM J. of Computing* **16**, No. 4 (1987), 760-778. [This is the successor to the Relational Queries paper that sets out much of the area of 'descriptive complexity.]

[I89] N. Immerman, "Descriptive and Computational Complexity," in *Computational Complexity Theory,* ed. J. Hartmanis, *Proc. Symp. in Applied Math.,* 38, American Mathematical Society (1989), 75-91.

[IL89] N. Immerman, S. Landau, "The Complexity of Iterated Multiplication," *Fourth Annual Structure in Complexity Theory Symp.* (1989), 104-111. To appear in *Information and Computation.*

[S*92] D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, S. Alagic. "Type-safe linguistic reflection", *Research report CS/92/6,* Department of Mathematical and Computational Sciences, University of St. Andrews, North Haugh, Scotland, 1992.

[SM*93] D. Stemple, R. Morrison, G.N.C. Kirby, R.C.H. Connor. "Integration, reflection, strong typing and static checking", *Research report CS/93/6,* Department of Mathematical and Computational Sciences, University of St. Andrews, North Haugh, Scotland, 1993.

[S91] I. A. Stewart, "Complete Problems Involving Boolean Labelled Structures and Projection Translations," *Journal of Logic Computation,* Vol. 1 No. 6, (1991), 861-882.

[S74] L. J. Stockmeyer [1974]. "The complexity of decision problems in automata theory and logic," MAC TR-133, Project MAC, MIT, Cambrige, Mass.

[U] J. Ullman, Principles of Database Systems, Computer Science Press.

[VVV93] J. Van den Bussche, D. Van Gucht, and G. Vossen, "Reflective Programming in the Relational Algebra," *Arbeitsgruppe Informatik, Universitat Giessen, Bericht Nr. 9210,* December 1992. Appeared in the Proceedings of the ACM Symposium on PODS, May, 1993.

[Var] M. Vardi, "Complexity of Relational Query Languages," *14th Symposium on Theory of Computation,* 1982, (137-146).