JOURNAL OF COMPUTER AND SYSTEM SCIENCES 47, 121–156 (1993)

Algebraic and Calculus Query Languages for Recursively Typed Complex Objects*

RICHARD HULL†

Computer Science Department, University of Southern California, Los Angeles, California 90089-0782

AND

JIANWEN SU‡

Department of Computer Science, University of California at Santa Barbara, Santa Barbara, California 93106

Received October 1989; accepted May 21, 1992

Algebraic and calculus database query languages for recursively typed complex objects based on the set and tuple constructs are studied. A fundamental characteristic of such complex objects is that, in them, sets may contain members with arbitrarily deep nesting of tuple and/or set constructs. Relative to mappings from flat relations to flat relations, the algebra without **while** has the expressive power of the algebra on conventional complex objects with non-recursive types. The algebra plus **while** has the power of the computable queries. The calculus has power equivalent to the arithmetical hierarchy and also to the calculus with countable invention for conventional complex objects. A technical tool, called "domain Turing machine," is introduced and applied to characterize the expressive power of several classes of relational queries.

1. INTRODUCTION

Since the relational data model [Cod70] has its significant drawbacks in many application areas (such as engineering design, CAD), various attempts have been made towards extending the model and its query languages (calculus and algebra)

* This work was supported in part by NSF Grant IRI-87-19875. A preliminary version of this work appeared as part of "Untyped Sets, Invention, and Computable Queries" in the *Proceedings of ACM Symp. on Principles of Database Systems, 1989*, pp. 347–359.

† This author was also supported in part by the Defense Advanced Research Projects Agency under NASA-Ames Cooperative Agreement No. NCC 2-520. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of NSF, DARPA, the U.S. Government, or any other person or agency connected with them.

‡ Work by this author was done primarily while the author was at the University of Southern California.

0022-0000/93 $5.00

[HK87]. One widely accepted approach focuses on so-called "complex objects," whose structure is formed using the *tuple* and *set* constructs. Within this approach two fundamentally distinct philosophies concerning the particular structures permitted of complex objects have emerged in the data base literature: (1) *Non-recursive types*, in which case all elements in the domain of a given type have *bounded nesting depth* of the tuple and set constructs. Examples include the nested relation and several complex object models which support homogeneous sets [Hul87, AB88, AG91, Kup87] and also complex object models supporting finitely heterogeneous sets, as in COL [AG91]. (2) *Recursive types*, which permit elements in the domain of a given type to have *arbitrarily deep nesting depth* of the constructs. This approach was pioneered in Database Logic [Jac82] and used more recently in a number of investigations, including, e.g., LDL [NT89], FAD [BBKV87], the deductive calculus of Bancilhon and Khoshafian [BK89], and also the "Set Theoretic Data Model" of Gemstone [CM84, MSOP86]. In this paper we conduct a theoretical study of expressive power of algebraic and calculus query languages for complex objects using recursive types. (A companion paper [HS93] explores analogous issues for deductive query languages.) Our results indicate that recursive types generally increase the power of a language considerably: in particular, the algebra with iteration has the power of computable queries [CH80]; and the calculus has the power of the arithmetical hierarchy [Rog87] (relativized to generic queries).

The investigation described here provides a bridge between research on the use of non-recursive complex objects in query languages [AB88, HS88, KV88, PvG88, GvG88], and research on computable queries [CH80, AV87]. While the complex object and nested relation models where developed several years ago, it is only recently that theoretical investigations of the expressive power and complexity of their associated query languages have been made. For example, the independent investigations [HS88, KV88] show that each level of nested sets gives more expressive power at an exponential cost (of time and space); and [AB88, GvG88] show that in the context of the nested relation algebra, the powerset construct is equivalent to various iterative constructs (e.g., fixpoint, **while**). On the other side, [AV87] introduced a transaction language which has the expressive power of (essentially) the computable queries originally introduced in [CH80].

In the investigation of calculus and algebraic languages here we restrict our attention to their behavior on flat relational input and output. In the context of non-recursive types, these languages have the same expressive power, that is, the expressive power of the class $\mathscr{E}$ of *elementary queries* [HS88] (see also [KV88]) which are generic mappings from flat relations to flat relations and computable by Turing machines within hyper-exponential time (and/or space).

In the presence of recursive types, it is shown that the algebra (with the powerset operator and without an iteration construct) is expressively equivalent to the one with non-recursive types. We also study the algebra extended by a **while** construct; in the presence of recursive types this is essentially subsumed by FAD. We show that every computable query is expressible in this extended algebra, regardless of

the presence of the powerset operator. Thus, recursive types break the "balance" between powerset and iteration [AB88, GvG88].

A second focus of the paper is the relationship between recursive types and the calculus. It is known [Var83] that the calculus of Database Logic can express non-computable queries. Intuitively, this follows from the fact that quantifiers in this calculus can range over infinite sets of structured objects, e.g., the set of all initial segments of a Turing machine computation. This permits the expression of conditions such as "there is no halting computation of Turing machine M on the empty string." In particular, the result that in the context of recursive types, the calculus is more powerful than the algebra (with or without iteration), provides an interesting contrast to the usual cases where the algebra and calculus over a given model have equivalent expressive power. This paper refines our understanding of the impact of recursive types on the calculus along two dimensions. First, it is shown that with recursive types, the calculus has expressive power equivalent to the arithmetical hierarchy [Rog87]. Second, it is shown here that the calculus with recursive types can be used to simulate "invented values" [AV87, HS88]; in particular, the calculus with recursive types has the same expressive power as the calculus for non-recursive types extended with countable invention.

In the formal development of the paper, we focus on a very general family of recursively defined types for complex objects, called *r-types*. Essentially any recursive complex object type is subsumed by some r-type; our characterizations thus provide upper bounds on the expressive power of languages supporting essentially arbitrary frameworks for recursively typed complex objects. On the other hand, all of the characterizations of expressive power are robust in the sense that they continue to hold for all recursive complex object typing disciplines discussed in the database literature.

In this paper we introduce a technical vehicle, called "domain Turing machine," which is useful for proving results about the expressive power of query languages. Unlike conventional Turing machines, domain Turing machines use an infinite input alphabet which corresponds to the underlying domain of (all) database instances considered. The transition function of domain Turing machines is finitely expressible and captures some of the spirit of the notion of "generic" queries. However, as with the use of conventional Turing machines to describe query functions, genericity is guaranteed here by focusing on domain Turing machines whose output is independent of the order of the input. Several classes of relational queries defined by complexity can be characterized in a natural fashion using domain Turing machines.

Section 2 reviews basic concepts and presents the specific algebra with non-recursive complex object types used in this paper. The section also reviews the conventional framework for using Turing machines in connection with query languages, to provide a basis for the discussion in Section 3, where domain Turing machines are introduced and studied. Section 4 presents a formalism for recursively typed complex objects, generalizes the conventional algebra to this context, and analyzes several variants of it. The recursively typed calculus is analyzed in

Section 5. Concluding remarks are included in Section 6. This paper (and its companion [HS93]) are based on portions of [HS89b]; as noted below, in some cases the exposition of technical details here is an abridged version of their exposition in [HS89b].

2. Preliminaries

In this section, we establish terminology for previously studied concepts including conventional complex objects and their types (non-recursive), databases, query functions, and the notion of equivalent expressive power. We introduce a specific complex object algebra for these non-recursively typed objects in this section to provide the basis for the recursively typed version of the algebra in Section 4. These algebras are based on assignment statements, to permit easy incorporation of while loops; in the non-recursive case it is (essentially) equivalent to other complex object algebras with powerset (e.g., [AB88]). We briefly review the standard calculus for complex objects here (e.g., [AB88, HS88]). The model will be extended to incorporate recursive types in Section 4. We also review the conventional framework for using Turing machines in connection with database queries, to provide a foundation for the introduction of domain Turing machines in Section 3. A variety of well-known classes of queries are introduced in the current section; the arithmetical hierarchy is introduced in Section 5.

We assume that **U** (the *universal domain*) and **P** are two disjoint countably infinite sets of *atomic objects* and abstract *predicate names* (respectively). The family of non-recursive types is defined recursively from the symbol "U" and the tuple and set constructs:

Definition. The set of *co-types* is a family of expressions defined recursively as follows:

(a) the symbol U is the *basic* co-type;

(b) $\{T\}$ is a *set* co-type if T is a co-type; and

(c) $[T_1, ..., T_n]$ is a *tuple* co-type if $n \geqslant 1$ and $T_1, ..., T_n$ are co-types.

A co-type is *flat* if it has the form $[U, ..., U]$.

Definition. The *domain* of a co-type T is denoted by $\text{dom}(T)$. The domains of co-types are defined recursively as follows:

(a) $\text{dom}(U) = \mathbf{U}$;

(b) $\text{dom}(\{T\}) = \mathscr{P}^{\text{fin}}(\text{dom}(T)) = \{Y \mid Y \subseteq \text{dom}(T) \text{ and } Y \text{ finite}\}$;

(c) $\text{dom}([T_1, ..., T_n]) = \text{dom}(T_1) \times \cdots \times \text{dom}(T_n)$.

An *object* of co-type T is an element of $\text{dom}(T)$; an *instance* of co-type T is a finite subset of $\text{dom}(T)$; and the family of instances of T is denoted $\text{inst}(T)$.

With the above definitions, we now define database schemas and instances.

DEFINITION. A *database schema* is a sequence $D = \langle P_1 : T_1, ..., P_n : T_n \rangle$, where

(a) $P_i \in \mathbf{P}$ for $i \in [1...n]$;

(b) T_i is a co-type for $i \in [1...n]$; and

(c) $P_i \neq P_j$ if $i \neq j$.

D is *flat* if T_i is flat for each $i \in [1...n]$. A (*database*) *instance* of D is a sequence $d = \langle P_1 : I_1, ..., P_n : I_n \rangle$, where I_j is an instance of T_j for each $j \in [1...n]$. The family of instances of D is denoted inst(D).

Finally, we define the dual notions of "active domain" and "constructed domain." The former captures the set of atomic objects used in building an object (instance of a co-type, database schema), and the latter identifies all objects built using a given set of atomic objects.

DEFINITION. For a co-type T and object $o \in \text{dom}(T)$, the *active domain* of o, denoted adom(o), is defined recursively as:

(a) if $T = U$ then $\text{adom}(o) = \{o\}$;

(b) if $T = \{T_1\}$ then $\text{adom}(o) = \bigcup \{\text{adom}(o') \mid o' \in o\}$; and

(c) if $T = [T_1, ..., T_n]$ and $o = [o_1, ..., o_n]$ then $\text{adom}(o) = \bigcup \{\text{adom}(o_i) \mid i \in [1...n]\}$.

The *active domain* of an instance I of T is $\text{adom}(I) = \bigcup \{\text{adom}(o) \mid o \in I\}$. The *active domain* of a database instance $d = \langle P_1 : I_1, ..., P_n : I_n \rangle$ is $\text{adom}(d) = \bigcup \{\text{adom}(I_j) \mid j \in [1...n]\}$. For a finite subset $X \subseteq \mathbf{U}$ and co-type T, the *constructed domain* of T using X is $\text{const}_T(X) = \{o \in \text{dom}(T) \mid \text{adom}(o) \subseteq X\}$.

In this paper, database queries are viewed as functions or mappings from database instances to instances of co-types. Now let D be a database schema and T be a co-type. A *database mapping* f from D to T, denoted $f: D \to T$, is a partial mapping from inst(D) into inst(T). Further let $d \in \text{inst}(D)$. We define $\text{indom}(f, d) = \text{adom}(d)$, and $\text{outdom}(f, d) = \text{adom}(f(d))$ ($\varnothing$ if $f(d)$ is undefined).

In principle, database queries treat data objects in an uninterpreted way. The functions, hence, are required to be "generic." Also, queries often have the property of not generating new values. These are formally defined as follows.

DEFINITION. *If* $C \subseteq \mathbf{U}$ is a finite set, f is a database mapping from D, then: f is

(a) (*input*) *domain preserving with respect to* C, if for every $d \in \text{inst}(D)$, $\text{outdom}(f, d) \subseteq \text{indom}(f, d) \cup C$;

(b) *C-generic* if $f \cdot \rho = \rho \cdot f$ for each permutation[1] ρ over $\mathbf{U}$ which leaves C fixed (i.e., $\forall x \in C$, $\rho(x) = x$).

[1] ρ is extended naturally to databases.

f is *domain preserving* (*generic*) if f is domain preserving with respect to C (C-generic) for some finite C.

It is easily verified that each generic database mapping is domain preserving. All queries in the languages discussed here are generic and domain preserving. For our purpose of investigation, we focus on those functions whose domains are flat databases (instances of flat database schemas) and ranges are flat relations (instances of flat co-types).

DEFINITION. A database mapping $f: D \to T$ is a *query function* if f is generic, D is a flat database schema, and T is a flat co-type.

We now introduce two interesting classes of queries: "computable queries" and "elementary queries." The family of "computable queries" was introduced in [CH80] and also studied in [AV87] and focuses on the class of generic database mappings which are Turing-computable. The different authors have used different classes of input and output schemas, and [AV90] studied both nondeterministic and deterministic mappings. In our study we use:

DEFINITION. The class $\mathscr{C}$ of *computable query functions* is the set of database mappings f such that f is a query function and f is Turing computable.

Because we shall introduce "domain Turing machines" in the next section, we now establish specific conventions for using Turing machines to define query functions.

Briefly, a (deterministic) *Turing machine* (TM) is a sextuple $M = (K, \Sigma, \Gamma, \delta, q_s, q_h)$; and an *instantaneous description* (ID) of M is a quadruple $(q, \alpha, \sigma, \beta)$, where q is the current state, σ is the symbol in the tape square where the head is, α and β are two strings of symbols representing the left and right part of the tape relative to the head. In general, Turing machines (1) have finite alphabets, (2) may compute non-generic functions, and (3) use ordered inputs (tape squares are ordered); this implies that an encoding of instances into a finite alphabet must be established and that not all Turing machines can be used. Here we describe a particular framework by which Turing machines are used to compute query functions. We assume for simplicity that the set Σ of symbols for Turing machines is the set $\{0, 1, ,, (,), [,]\}$, and Σ is disjoint from $\mathbf{U}$. A mapping μ from[2] $\mathbf{U}$ to $\{0, 1\}^*$ is an *encoding* of $\mathbf{U}$ if μ is one-to-one and onto. μ is extended naturally to an one-to-one and onto mapping from $\mathbf{U} \cup \{,, (,), [,]\}$ to $\{0, 1\}^* \cup \{,, (,), [,]\}$ which is an identity mapping on $\{,, (,), [,]\}$. Let μ^{-1} denote the inverse of μ.

Now let D be a flat database scheme, T a flat co-type, $f: D \to T$ a C-generic query function where $C \subset \mathbf{U}$ is finite and d is an instance of D. Since relations in d are sets of tuples without ordering while the input for Turing machines must be a sequence, we specify this transformation briefly as follows: First, an enumeration of d is a

[2] For any alphabet Σ, Σ^* is the set of strings over Σ.

sequence e, which lists all tuples in d, where the coordinate values of tuples are separated by ","s, where tuples are delimited by "(" and ")"; and relations are delimited by "[" and "]". (Relations are listed in the order given be the schema D.) Let μ be an encoding of **U**. We further extend μ to a mapping from enumerations to Σ^* in the natural fashion. Since μ is identity on punctuation symbols $(\Sigma - \{0, 1\})$ its inverse can also be defined to map strings back to enumerations. Finally, the input for Turing machines is $\mu(e)$.

EXAMPLE 2.1. Let $D = \langle R_1 : [U, U], R_2 : [U] \rangle$ be a database schema and $d = \langle R_1 : \{[a, b], [b, c]\}, R_2 : \{[a], [b], [c]\} \rangle$ an instance of D. Then $e =$ [(a, b)(b, c)][(a)(b)(c)] is an enumeration of d. Further, if μ is an encoding with $\mu(a) = 00$, $\mu(b) = 01$, $\mu(c) = 10$, $\mu(e) =$ [(00, 01)(01, 10)] [(00)(01)(10)].

A TM M is *input-order independent* with respect to input database schema D, output co-type T (and encoding μ), if for every instance d of D, either (a) for every enumeration e of d, M does not halt; or (b) there is an instance I of T such that for every enumeration e of d, M halts and the output of M, denoted $M\mu(e)$, is an encoded sequence satisfying the condition that $\mu^{-1}M\mu(e)$ is some enumeration of I. Now, a Turing machine M *computes* the query function f from D to T (using encoding μ) if M is input-order independent with respect to D, T, and μ, and for each instance d of D, either $f(d)$ is defined and M halts with output an encoded enumeration of the relation $f(d)$, or $f(d)$ is undefined and M does not halt on encoded enumerations of d. We include the following straightforward observation which will be used in discussions on domain Turing machines in the Section 3. Intuitively, the lemma states that the encoding is transparent to M up to C.

LEMMA 2.2. *Suppose that $f: D \to T$ is a C-generic query function and d is an instance of D. Further let μ, ν be two encodings of* **U**, *such that for each $a \in C$, $\mu(a) = \nu(a)$. If M is a Turing machine which computes f using μ, then M also computes f using ν; i.e.,*

$$\mu^{-1}M\mu(e) \quad \text{and} \quad \nu^{-1}M\nu(e)$$

are two enumerations of $f(d)$ whenever $f(d)$ is defined.

Proof. Suppose that $\rho = \mu^{-1}\nu$. Then for each $a \in C$, $\rho(a) = a$, since $\mu(a) = \nu(a)$. Hence, ρ is a permutation of **U** which leaves C fixed. Now consider $\nu^{-1}M\nu(e)$ which equals $\rho^{-1}\mu^{-1}M\mu\rho(e)$. Since M computes f using μ, $\mu^{-1}M\mu\rho(e)$ and $\rho\mu^{-1}M\mu(e)$ are enumerations of $f(\rho(d))$ and $\rho(f(d))$, respectively, if defined (the orderings of tuples may differ). By C-genericity of f, $f(\rho(d)) = \rho(f(d))$. Applying ρ^{-1} to these, we conclude that $\mu^{-1}M\mu(e)$ and $\nu^{-1}M\nu(e)$ represent the same $f(d)$ up to the ordering of tuples. ∎

Another interesting and important aspect of studying query functions concerns complexity issues. In this realm, the focus is typically on total query functions. Also,

Turing machines are used to measure the complexity in a way different from the "computing" mode just described, they instead use a "testing" mode, which we now sketch.

Let $f: D \to T$ be a (total) query function, d an instance of D, and o an object of co-type T. A Turing machine M under the testing mode will have the input "$\mu(e)\,\mu(o)$," where e is an enumeration of d and μ an encoding of **U**. M may halt with an output of a single `1` (Yes, $o \in f(d)$) or `0` (No, $o \notin f(d)$). If $\sigma: \mathbf{N} \to \mathbf{N}$ is a total function, M *tests* f (within time (space) σ) if M is input-order independent and halts for all inputs; given any database d, for each encoded enumeration e and each encoded object o, M produces output consistent with f within[1] time (space) $\sigma(\|\mu(e)\| + \|\mu(o)\|)$.

DEFINITION. Let $f: D \to T$ be a query function and $\sigma: \mathbf{N} \to \mathbf{N}$ be a function. f has *time (space) complexity* σ, if there exists a Turing machine M which tests f within time (space) $\sigma(\|\mu(e)\| + \|\mu(o)\|)$ whenever d is an instance of D, e is an enumeration of d, μ is an encoding of **U**, and o is an object of the co-type T.

The notion of "elementary queries" arises naturally in the query languages for complex objects (see Theorem 2.5). Roughly speaking, this family contains queries whose time (space) complexity are elementary functions.

Notation. The class of *hyper-exponential* functions hyp_i ($i \in \mathbf{N}$) is defined such that $\text{hyp}_0(n) = n$ and $\text{hyp}_{i+1}(n) = 2^{\text{hyp}_i(n)}$, for $i \geqslant 0$.

DEFINITION. The class $\mathscr{E}$ (QPTIME, QPSPACE, QLOGSPACE) of *elementary* (*polynomial time*, *polynomial space*, *logspace*) query functions is the set of query functions f such that:

(a) $f \in \mathscr{C}$; and

(b) f has hyper-exponential time/space (polynomial time, polynomial space, logspace[4]) complexity.

On the other hand, we can also discuss the complexity of Turing machines which run in computing mode. For example, the elementary class can be defined alternatively as:

DEFINITION'. The class $\mathscr{E}$ (QPTIME, QPSPACE) of *elementary* (*polynomial time*, *polynomial space*) query functions is the set of functions f:

(a) $f \in \mathscr{C}$; and

(b) f can be computed by some Turing machine in hyper-exponential time/space (polynomial time, polynomial space).

[3] $\|\mu(e)\|$ denotes the length of $\mu(e)$.

[4] In this case we use TMs with separate read and work tapes.

Since we focus here on flat schemas and flat co-types, the number of possible objects to be tested is polynomially many. As noted in [CH82], if F is a class of functions including identity and closed under taking polynomials, then there is a Turing machine testing a given query function f within time (space) $g \in F$ if and only if there is a Turing machine which computes f and has running time (space) bounded by some $g' \in F$.

The following proposition is easily verified.

PROPOSITION 2.3. *The two definitions of* $\mathscr{C}$ (QPTIME, QPSPACE) *yield the same class of query functions.*

A query is syntactically an expression. With associated semantics, each query expression *realizes* a generic database mapping. As our focus is on query functions which are from flat schemas to flat co-types, we consider only query expressions with that property in the rest of the paper.

The algebra used here is essentially equivalent to those of [AB88, KV84, RKS88]. Assume that **V** is a disjoint set of *variables*. We use a syntax which views algebra expressions as sequences of assignments, each of which applies a single operator (e.g., in the spirit of [KV84]). This permits the easy incorporation of the **while** construct into the algebra (in the spirit of [GvG88]). In our framework, variables have associated co-types. In particular, a variable of co-type T ranges over inst(T), i.e., all instances of T.

DEFINITION. A *co-type assignment* τ is a (total) mapping from $\mathbf{P} \cup \mathbf{V}$ to the set of co-types.

DEFINITION. Let τ be a co-type assignment and x, y, z variables in **V**. A (*co-typed*) *statement* is of one of the following form:

(1) $x := y$, if $\tau(x) = \tau(y)$;

(2) $x := P$, if $P \in \mathbf{P}$ and $\tau(x) = \tau(P)$;

(3) $x := \{a\}$, if $a \in \mathbf{U}$ and $\tau(x) = U$;

(4) $x := y \cup z$ (or $y \cap z$, $y - z$), if $\tau(x) = \tau(y) = \tau(z)$;

(5) $x := \pi_{i_1, \ldots, i_k}(y)$, if $\tau(y) = [T_1, \ldots, T_n]$, $i_1, \ldots, i_k \in [1 \ldots n]$, and $\tau(x) = [T_{i_1}, \ldots, T_{i_k}]$;

(6) $x = \sigma_F(y)$, if $\tau(x) = \tau(y) = [T_1, \ldots, T_n]$ and F is a *selection formula* of form:

(a) *atomic*: $t_1 = t_2$ or $t_1 \in t_2$, where for $i = 1, 2$, $t_i =$ "a" for some $a \in \mathbf{U}$, or $t_i \in [1 \ldots n]$ (where the co-types of t_1 and t_2 are correct and consistent with the predicate);

(b) a formula built using sentential connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$) from atoms;

(7) $x := y \times z$, if $\tau(x) = [\tau(y), \tau(z)]$;

(8) $x :=$ untuple(y), if $\tau(y) = [\tau(x)]$;

(9) $x := \text{aggregate}_k(y)$, if $\tau(y) = [T_1, ..., T_n]$, $k \leqslant n$, and $\tau(x) = [[T_1, ..., T_k], ..., T_n]$;

(10) $x := \text{disaggregate}(y)$, if $\tau(y) = [[S_1, ..., S_m], T_1, ..., T_n]$ and $\tau(x) = [S_1, ..., S_m, T_1, ..., T_n]$;

(11) $x := \text{nest}(y)$, if $\tau(y) = [T_1, T_2, ..., T_n]$ and $\tau(x) = [\{T_1\}, T_2, ..., T_n]$;

(12) $x := \text{unnest}(y)$, if $\tau(y) = [\{T_1\}, T_2, ..., T_n]$ and $\tau(x) = [T_1, T_2, ..., T_n]$

(13) $x := \text{collapse}(y)$, if $\tau(y) = \{\tau(x)\}$;

(14) $x := \text{powerset}(y)$, if $\tau(x) = \{\tau(y)\}$; or

(15) a **while** statement with the following format:

while x **do** $S_1; ...; S_n$; **end**

where x is a variable and $S_1, ..., S_n$ are statements. The **while** statement is *unnested* if none of $S_1, ..., S_n$ are **while** statements; otherwise it is *nested.*

We briefly describe the *semantics* for the statements. *Untuple* deprives objects of the top level tuple construct; the *cross-product* operator always returns ordered pairs; *aggregate* (*disaggregate*) constructs (destructs) tuples nested within a tuple construct, which do not appear in nested relations; *nest* and *unnest* operators[5] are simplified to operate on only the first column. The **while** statement has the semantics: while the value of x is defined but not empty execute the assignments; if x is undefined, the result of the **while** statement (and hence the query) is undefined. The remaining operators have the usual semantics.

The algebra presented above is slightly more powerful than the nested algebra (which contains the relational operators, nest, unnest, and powerset) since the tuple construct can be nested in this context; but is equivalent to the algebra of [AB88]. In the latter, a complex "replace" operator is used instead of a set of operators (e.g., project, select).

DEFINITION. Let D be a flat database schema and T a flat co-type. An *algebraic query expression* Q from D to T, denoted as $Q: D \to T$, is a sequence of statements $S_1, ..., S_n$ followed by an assignment (non-**while** statement) to a special variable ANS of co-type T such that all predicate name used in $S_1, ..., S_n$ are in D and each variable is assigned a value before it is referenced. Let $^{co}\text{ALG} + \textbf{while}$ ("co" for complex objects) denote the family of all such query expressions. Also, we denote ^{co}ALG ($^{co}\text{ALG} + \text{unnested-}\textbf{while}$) as the family of algebraic query expressions without (nested) **while** (respectively).

If Q is an algebraic query expression from D to T and d is an instance of D, the *semantics* (*answer*) of Q on d, denoted $Q[d]$, is defined in a usual fashion with the

[5] The "nest" and "unnest" operators are redundant. They are included for convenience in expressing queries.

value assigned to ANS as the output, where the query evaluates to the undefined value (denoted as "$\perp$") if a **while** loop does not terminate.

The calculus for non-recursive types we use is that of [HS88] and is similar to those of [AB88, KV84]. In the calculus, *terms* include variables, $t.i$, where $t.i$, where t is a variable and $i \in \mathbf{N}$, and elements of **U**. *Formulas* are built from $s = t$, $s \in t$, $P(t)$ (P is a predicate name, s and t are terms) using sentential connectives ($\wedge$, $\vee$, $\neg$) and typed quantifications ($\forall x/T\phi$, $\exists x/T\phi$) where T is a co-type. A calculus query expression is an expression of form $\{t/T \mid \phi\}$, where ϕ is a (well-typed) formula. The semantics of an expression is defined by taking the set of all atomic objects occurring in the database or the expression as the universe. This semantics is also called "limited interpretation" or no invention (of new atomic objects). Let ^{co}CALC represent the family of all calculus query expressions. We also discuss two other semantics in Section 5: finite invention and countable invention (unlimited interpretation).

DEFINITION. Two query languages L_1 and L_2 are *equivalent* if they realize the same set of query functions. L_1 is no more expressive than L_2 ($L_1 \sqsubseteq L_2$) if each query function realizable in L_1 is also realizable in L_2. If S is a family of query functions, L_1 is *S-equivalent* if

(a) every query expression realizes some f in S, and

(b) every query function in S is realizable by L_1.

Thus, the notion of $\mathscr{C}$-equivalent has been called "computationally complete" in other investigations. When the context is clear, both query functions and query expressions are referred as simply *queries.*

The expressive power of many (co-typed) complex object languages is characterized by the following two theorems. The first theorem shows that the various languages for complex objects are essentially equivalent in expressive power [AB88, GvG88].

THEOREM 2.4. ^{co}ALG, ^{co}ALG + (unnested)-**while**, COL [AG91], ^{co}CALC, *and the recursive language of* [AB88] *are all equivalent.*

The second states that the languages capture elementary query functions [HS88] (see also [KV88]).

THEOREM 2.5. ^{co}CALC *is* $\mathscr{E}$*-equivalent. Hence,* ^{co}ALG, ^{co}ALG + (unnested)-**while**, COL, *and the recursive language of* [AB88] *are also* $\mathscr{E}$*-equivalent.*

3. DOMAIN TURING MACHINES

This section introduces and studies "domain Turing machines" (domTMs), a variant of Turing machines which is focused on database manipulation. Unlike

conventional Turing machines, the tape alphabet of a domTM includes the universal domain U, an infinite set. The transition function of a domTM is finitely expressed, and is essentially "generic." However, the correspondence between domTMs and queries is not complete; a conventional Turing tape is used for input and processing, and so the mappings computed and/or tested by domTMs may depend on input order. For this reason, we restrict attention here to input-order independent domTMs[6].

Domain Turing machines can be used to simplify proofs that query languages are $\mathscr{C}$-equivalent. In the previous literature such proofs have generally been accomplished by showing that a given query language can simulate an arbitrary Turing machine (TM) which computes a generic query function. The use of domTMs permits the elimination of the conceptual steps of encoding domain elements into strings over a finite alphabet and subsequently decoding them. This approach is used in the present paper and its companion [HS93], and also in [She90]. (The use of complete relational query languages such as QL [CH80], detTL [AV87], or detDL [AV88] provides another approach to such proofs.)

A useful feature of domTMs stems from the natural relationship between the running time and space requirements of domTMs and conventional TMs and conventional TMs. This permits natural characterizations of query families such as QLOGSPACE, QPTIME, QPSPACE, $\mathscr{E}$, and $\mathscr{C}$ (see Corollary 3.4 and Proposition 3.5). It would be interesting to develop analogous characterizations for these language families in terms complete relational query languages.

We begin with some intuitive remarks. Consider the problem of showing that a given database query language can simulate an arbitrary Turing machine which computes a generic query function, as in, e.g., [CH80, AV87]. Three fundamental issue arise:

1. the tape alphabet of the Turing machine is finite, but the underlying domain of input instances is infinite.
2. the computation of the Turing machine must be independent of the order in which the input instance is presented to it.
3. the Turing machine must be restricted so that it computes (tests) a generic mapping.

The notion of domain Turing machine focuses primarily on the first and third issues. A domTM can directly manipulate infinitely many tape symbols, including all atomic objects in U. As with conventional TMs used to define query functions, we shall restrict our attention to domTMs which are input-order independent. Under that assumption, the definition of domTM will ensure that the computed (tested) function is generic.

Intuitively, a domTM M is a Turing machine with a two-way infinite tape and a *register*, which can be used to store a single letter of the alphabet. Unlike conven-

[6] S. Abiteboul and V. Vianu have recently developed [AV91] a Turing-machine based automaton for database queries which overcomes the need for the restriction to input-order independence.

tional TMs, the alphabet of a domTM includes the underlying domain **U** and a finite set W of *working* symbols. (This will include the punctuation symbols [,], (,) and ,; and also ♭.) Also, a finite set $C \subset \mathbf{U}$ of constants is explicitly specified in M—these correspond to the constants used in a query, and the computation of M will be C-generic. The transition function for M explicitly uses the members of $W \cup C$ to refer to tape symbols, and it also uses the distinguished variables η and κ, which are used to refer to elements of $\mathbf{U} - C$.

We now present the formal definition of the domTMs.

DEFINITION. A (*deterministic*) *domain Turing machine* (domTM) (relative to **U**) is a sextuple

$$M = (K, W, C, \delta, q_s, q_h),$$

where

1. K is a finite set of *states.*
2. W is a finite set of *working symbols.* (In the current discussion we assume that W includes the distinguished symbols "`,`, `(`, `)`, `[`, `]`, `0`, `1`" which are used for encoding input and output, and also ♭.)
3. $C \subset \mathbf{U}$ is a finite set of *constants.*
4. $q_s \in K$ is the *start state.*
5. $q_h \in K$ is the unique *halting state.*
6. δ is the *transition function* from $(K - \{h\}) \times (W \cup C \cup \{\eta\}) \times (W \cup C \cup \{\eta, \kappa\})$ to $K \times (W \cup C \cup \{\eta, \kappa\})^2 \times \{\mathrm{L}, \mathrm{R}, —\}$. In a transition value $\delta(q, a, b) = (q', a', b', dir)$, $b = \kappa$ only if $a = \eta$; $\eta \in \{a', b'\}$ only if $\eta \in \{a, b\}$; and $\kappa \in \{a', b'\}$ only if $\kappa \in \{a, b\}$.

M is viewed as having a two-way infinite tape and a *register.* An *instanteneous description* (ID) of M is a five-tuple (q, a, α, b, β), where q is a state; $a \in W \cup \mathbf{U} \cup \{♭\}$ (the *register contents*); α, $\beta \in (W \cup \mathbf{U})^*$ and $b \in W \cup \mathbf{U}$ (the *tape contents* are $\alpha b \beta$, where the *tape head position* is the specified occurrence of b). (We assume the usual restriction that neither the first symbol of α nor the last symbol of β is ♭.) A transition value $\delta(q, a, b) = (q', a', b', dir)$ is *generic* if $\eta \in \{a, b\}$. Intuitively, a generic transition value is used as a template for an infinite set of transition values which are formed by letting η (and κ if it occurs) range over (distinct) elements of $\mathbf{U} - C$. At the beginning of a computation the register holds ♭. Under these provisions, a *computation* of M is defined in the usual fashion.

It is easily verified that the restrictions on δ in the above definition ensure that the domTM M is deterministic. The following illustrates the basic definition of domTM.

EXAMPLE 3.1. We define (parts of) a domTM which computes $\sigma_{1=3}(R)$ of a ternary relation. This domTM will have no constants (i.e., $C = \varnothing$). Recall that the inputs will be encodings of ternary relations. The transition function δ includes

(1) $\delta(q_s, \not{b}, [) = (q_1, \not{b}, [, R)$
(2) $\delta(q_1, \not{b}, () = (q_1, \not{b}, (, R)$
(3) $\delta(q_1, \not{b}, \eta) = (q_2, \eta, \eta, R)$
(4) $\delta(q_2, \eta, ,) = (q_2, \eta, ,, R)$
(5) $\delta(q_2, \eta, \kappa) = (q_3, \eta, \kappa, R)$
(6) $\delta(q_2, \eta, \eta) = (q_3, \eta, \eta, R)$
(7) $\delta(q_3, \eta, ,) = (q_3, \eta, ,, R)$
(8) $\delta(q_3, \eta, \eta) = (q_1, \not{b}, \eta, R)$
(9) $\delta(q_3, \eta, \kappa) = (q_1, \not{b}, \not{b}, R)$
(10) $\delta(q_1, \not{b},)) = (q_1, \not{b},), R)$
(11) $\delta(q_1, \not{b},]) = (q_4, \not{b},], R)$.

Here transitions (1) and (2) get to the first atomic object of the input encoding. Transition (3) "remembers" that atomic object in the register. Transitions (4) through (7) skip over the second coordinate of the tuple. Transitions (8) and (9) compare the third coordinate of the tuple with the register contents; if they match transition (8) leaves the tuple unchanged, and if they do not match, transition (9) changes the third coordinate to $\not{b}$ so that it can be deleted later in the computation. Both of these transitions also replace the register contents by $\not{b}$. Transition (10) reads over the ")" and transfers control back to transition (2). The end of the input is detected by transition (11), which turns control over the state q_4. Although the details are not included here, the computation from state q_4 erases the tuples marked by a $\not{b}$ in the third coordinate and arranges the remaining tuples so that they are listed without separation.

The following lemma is easily verified using a straightforward induction.

LEMMA 3.2. *Let* $M = (K, C, W, \delta, q_s, q_h)$ *be a* domTM *and let* $\rho: \mathbf{U} \to \mathbf{U}$ *be an arbitrary permutation which leaves* C *fixed. Extend* ρ *to* W *by defining* $\rho(w) = w$ *for each* $w \in W$. *Let* $\alpha \in (W \cup \mathbf{U} - \{\not{b}\})^*$. *Then* M *halts on input* α *iff* M *halts on input* $\rho(\alpha)$. *Furthermore, if these computations halt then the output of the computation on* $\rho(\alpha)$ *is equal to* ρ *applied to the output of the computation on* α.

As in the case of using conventional TMs to define query functions, domTMs can be used in both *compute* mode and *test* mode. We introduce both of these now, beginning with compute mode. Let M be a domTM, D a flat database schema, and T a flat co-type. As input for M we use an enumeration e of an instance d of D. M is started on input e and computes until it reaches the halting state (if ever). If M halts and the contents of the tape hold an ordered listing of an instance of T, that instance is the output of the computation of M on input d with enumeration e. If M does not halt, or if the contents of the first tape is not an ordered listing of an instance of T, then M produces the undefined output on d with enumeration e. M is *input-order independent* from D to T if, for each instance d of D, the output

of M is the same (up to ordering of the tuples) regardless of the enumeration used for d. In this case we say that M *computes* the function f_M, where for each instance d of D, $f_M(d)$ is the instance corresponding to the output of the execution of M on some (any) enumeration e of d. Using Lemma 3.2 it is straightforward to verify that if a domTM M computes f_M, then f_M is generic, and so it is a query function.

The test mode for domTMs is the natural analog of test mode for TMs. In this mode we assume that the domTM halts on all inputs. Suppose that D and T are as before. Let e be an enumeration of an input instance d, and let t be a tuple with the same width as T. M is started on input et, and computes until it halts. If M halts with 1 as output, then M *accepts* t for input d with enumeration e; if M halts with 0 as output, then M *rejects* t for input d with enumeration e. M is *input-order independent* from D to T if, for each instance d of D and each tuple over T, the result of the computation of M is the same regardless of the enumeration used for d. In this case, M *tests* the function g_M defined so that

$$g_M(d) = \{t \mid g_M \text{ accepts } et \text{ for some (any) enumeration } e \text{ of } d\}.$$

Using Lemma 3.2 it is easily verified that if M is input-order independent then g_M is a query function. If $\tau: \mathbf{N} \to \mathbf{N}$ and M operates within time (space) τ, then M *tests* the function g_M in time (space) τ.

We emphasize here that in the cases of computing and testing with domTMs, the generic nature of domTM transition functions and the requirement of input-order independence together imply that the resulting mappings are generic. It is straightforward to show that it is undecidable, given a domTM, whether or not it is input-order independent.

It is easily seen that if the register is not included in the definition of domTM, then domTMs would have power strictly less than that of TMs (which compute query functions). For example, a domTM without a register cannot replicate a non-constant domain element or test the equality of two non-constant domain elements. Thus, for example, such domTMs cannot compute or test the query function $\sigma_{1=2}(R)$ for a binary relation R.

It is easy to define the notion of k-tape domTMs for $k \geqslant 2$. It can be shown that for $k \geqslant 2$, k-tape domTMs do not need a register. It is also straightforward to show that each k-tape domTM can be simulated by 1-tape domTM (with register).

We now turn to a proposition which shows that domTMs can simulate TMs in the context of database mappings and which establishes a number of natural correspondences between domTMs and TMs. This proposition permits the use of (input-order independent) domTMs rather than TMs in proofs, thus allowing us to avoid consideration of an encoding μ when simulating domTM computations within a query language. A corollary to the proposition characterizes a number of well-known complexity-based classes of queries.

At first glance it may seem counterintuitive that a domTM, whose transitions are restricted in a manner to capture genericity, should be able to simulate an arbitrary TM. A fundamental reason that domTMs can simulate TMs is that domTM inputs

are inherently ordered; domTMs can use this to provide an order of the atomic objects in their input. DomTMs which are not required to be input-order independent can thus "circumvent" the generic aspect of their transitions. This does not cause problems in our development, because we focus exclusively on input-order independent TMs and domTMs.

PROPOSITION 3.3. 1. *Suppose that M' is a* TM *which computes (tests) the query function f. Then there is a* domTM *M which computes (tests) f. Furthermore, if M' computes f in time τ and/or space σ (where both of these are assumed to be monotonically non-decreasing functions such that $\tau(n) \geq n$ and $\sigma(n) \geq n$ for each n), then M can be chosen to compute f in time $O(\tau(n \log n)^2)$ and/or space $O(\sigma(n \log n))$. If M' tests f in time τ and/or space σ (satisfying the same conditions), then M can be chosen to test f in time $O((n \log n)\, \tau(n \log n))$ and/or space $O(\sigma(n \log n)$.*

2. *Suppose that M is a* domTM *which computes (tests) the query function f. Then there is a* TM *M' which computes (tests) f. Furthermore, if M computes (tests) f in time τ and/or space σ (where both of these are assumed to be monotonically non-decreasing functions), then M' can be chosen to compute (test) f in time $O(n\tau(n)^2)$ and/or space $O(n\sigma(n))$.*

Proof. Let D be a database schema and T a relation schema. Suppose that the TM M' computes the C-generic database function f for some finite $C \subset \mathbf{U}$, using the encoding μ. A domTM M which simulates M' can be constructed as follows: M will use the set C as its constants and includes as working symbols $\{0, 1, 0', 1'\}$ and also the punctuation symbols \$, @, # the latter three symbols are assumed to be outside of the tape alphabet of M'). The computation of M has the following stages:

(a) Given an input e representing instance d on its tape, M creates a *dictionary* which holds a one–one mapping of $\text{adom}(d) \cup C$ into $\{0, 1\}^*$. The dictionary is stored as a sequence of pairs $a \,@\, x$, where $a \in \text{adom}(d) \cup C$ and $x \in \{0, 1\}^*$ is the encoding of a. The encoding is chosen so that c is encoded by $\mu(c)$ for each $c \in C$; and the elements of $\text{adom}(d) - C$ are assigned the lexicographically smallest members of $\{0, 1\}^* - \mu[C]$ possible. (Of course, the encodings used will depend on the particular ordering of d given by e.) The dictionary will be stored to the right of the input word. Note that the dictionary will have size at most $O(n \log n)$, where n is the size of the original input word e.

Briefly, the creation of the dictionary requires a number of *major steps*, each of which considers a symbol a in e, and either generates a new pair $a \,@\, x$, or determines that an encoding for a has already been established. (A more complete description of this stage is presented in [HS89b].)

(b) Let ν be the (partial) encoding of $\mathbf{U}$ into $\{0, 1\}^*$ defined by the dictionary. Note that for each $c \in C$, $\nu(c) = \mu(c)$. M now uses the dictionary to create the string $\nu(e)$. The string $\nu(e)$ will be created to the right of the dictionary, separated from it by the string \$\$. This can be accomplished by performing another sweep

through the original input, this time from left-to-right, again using the # as a place marker. The register is used heavily in this stage, and the symbols $\{0', 1'\}$ can be used when transferring a domain element encoding from the dictionary to the string $v(e)$. For efficiency in the third stage, the original input is erased during the sweep across it.

(c) M next simulates the operation of M' on the string u. This step is done without reference to the original input or the encoding. If the simulation of M' needs space to the left of its current working area (i.e., if M encounters a \$), then the dictionary is shifted to the left by one square. (This is called a *left-shift*.)

(d) Next, if the simulation of M' halts, then M uses the dictionary to "decode" output of the simulation (again using the register).

(e) Finally, M erases all of the extraneous symbols and then halts.

Under the assumption that M' computes the query function f (and is hence independent of input ordering and C-generic), it is straightforward to verify that the domTM M described above is input-order independent and computes. Also, Lemma 3.2 ensures that $f_M = f$.

We now consider the running time and space of the domTM M. Suppose first that M' has running time τ and/or space σ, for some monotonically non-decreasing function(s) τ and/or σ which are $\geqslant$ the identity. The first stage of M's operation takes time $O(n^2 \log n)$ and space $O(n \log n)$. (Here the $n \log n$ factor stems from the maximum possible length of words of $\{0, 1\}^*$ needed to encode elements of $\text{adom}(d) - C$, and the big-O stems from the inclusion of all elements of C in the dictionary). The second stage of M's operation takes time $O((n \log n)^2)$ and additional space $O(n \log n)$. The third stage of M's operation, the simulation of M', may involve many left-shifts of the dictionary. Each left-shift requires $O(n \log n)$ steps, and so this stage of M's operation takes time $O((n \log n)\, \tau(n \log n))$. The additional space needed for this stage is $O(\sigma(n \log n))$. (These estimates use the assumptions that τ and/or σ are monotonically non-decreasing, because the estimate of the length of the encoded input for M may be longer than its actual length.)

The fourth stage, for decoding the output, will take time $O((\text{length of output of } M')(\sigma(n) + n \log n)) \leqslant O(\tau(n)^2 + \tau(n)\, n \log n)$. No additional space is needed for this stage, because the encoded output of M' requires at least as much space as the decoded output of M. The final stage of erasing extraneous symbols will take no more than $O(\tau(n) + n \log n)$. No additional space is required for this stage, either. By the assumption that τ and/or σ are $\geqslant$ identity, it is now easily seen that the time and/or space bound(s) required of M are satisfied.

Consider now the case where M' tests the query function f. Essentially the same construction of M can be used, except that stages 4 and 5 are not needed. Furthermore, it is easily verified in this case that the stated time and/or space bounds are satisfied by M. This completes the proof of the first part of the proposition.

For the other direction, suppose that the C-generic query function $f: D \to T$ is computed by a domTM M. We briefly indicate how to construct a (conventional)

Turing machine M' which simulates M. M' takes as input some encoded enumeration $\mu(e)$ of database instance d. To simulate the behavior of M, M' first expands the input by placing the special symbol # between each ,, (,), [,] and any other symbol. As a result, each portion of the input in between #'s is either a single punctuation letter, or is $\mu(a)$ for some atomic object a occurring in e. M' will treat these subwords as single symbols for the purposes of simulating M. Note that the finite state control of M' can be arranged to be able to recognize elements of $\mu[C]$; other elements of $\{0, 1\}^*$ will be treated generically. Finally, M' will use the portion just to the left of everything else to hold (the encoding of) the contents of M's register during the simulation; this is separated from the other part of the tape by a \$ symbol.

The simulation of M by M' is now straightforward. Each step of M will typically require many steps of M': to check the contents of the simulated register; copy the contents of the register to a simulated tape location or visa versa; and to accomodate the variables lengths of the simulations of M's tape symbols.

Suppose now that M runs in time τ and/or space σ. The space needed for M' has order at most the product of the space needed for M on e $(=\sigma(\|e\|))$ plus room for the register $(=1)$ with the maximum length of a word in $\{0, 1\}^*$ corresponding to elements of $\text{adom}(d) \cup C$. More specifically, this is $(\sigma(\|e\|)+1) \cdot \|\mu(e)\| \leqslant \|\mu(e)\|\, \sigma(\|\mu(e)\|) \in O(n\sigma(n))$. (An improvement of this bound to $O(\log n\sigma(n))$ can be obtained by translating $\mu(e)$ to $\nu(e)$, where $\nu(c) = \mu(c)$ for each $c \in C$ and where $\nu(a)$ is chosen to be as short as possible for each $a \in \text{adom}(d) - C$.)

Suppose now that M runs in time τ. In the worst case, simulated steps of M' will involve traversing the length of the tape $\|\mu(a)\|$ times where a is the tape contents under the head or the register contents, in order to compare the tape contents with the register and/or to shuffle everything right or left to accomodate replacing the tape contents with a letter with an encoding of a different length. Thus, the time required is $\leqslant O(n \cdot \sigma(n) \cdot \tau(n)) \leqslant O(n\tau(n)^2)$. (Under the refinement suggested above there is a considerable time expense in setting up the correspondence between μ and ν; a bound on the total time used is $O(\log n\tau(n)^2 + \sigma(n)(\sigma(n) + n^2 \log n)) \leqslant O(\log n\tau(n)^2 + \tau(n)(\tau(n) + n^2 \log n))$.)

The same analysis can be used for the case of testing (except that under the refinement, the translation of the result can be omitted, deleting the $\sigma(n)^2$ term from the running time). ∎

COROLLARY 3.4. 1. *The family of mappings computed by* domTMs *is* $\mathscr{C}$.

2. *The family of mappings computed (tested) by* domTMs *running in hyper-exponential time (space) is* $\mathscr{E}$.

3. *The family of mappings computed (tested) by* domTMs *running in polynomial space is* QPSPACE.

4. *The family of mappings computed (tested) by* domTMs *running in polynomial time is* QPTIME.

The family QLOGSPACE of queries is of interest because the relational calculus

and algebra are known to be QLOGSPACE-complete in terms of data complexity [Var82]. We now present a natural variant of domTMs which permits an interesting characterization of QLOGSPACE. A *read-restricted* domTM is like a domTM, except that it is a read-only *input tape*; a read-write *work tape*; and as before, a single register. As with domTMs, a read-restricted domTM will use a distinguished set W of *working symbols* and a finite set $C \subseteq \mathbf{U}$ of *constants*. The input tape will contain domain elements and punctuation only; the register can contain arbitrary domain elements and working symbols of M; and the work tape can contain only members of $W \cup C$. Read-restricted domTMs define mappings only by testing members; and the notion of input-order independent is defined as before. The *space used* by a read-restricted domTM is determined by the amount of work tape space used.

The following proposition characterizes QLOGSPACE using read-restricted domTMs. The proof is tangential to the main theme of the paper and is included in an appendix.

PROPOSITION 3.5. *The family of query functions tested by read-restricted* domTMs *operating in logspace is* QLOGSPACE.

In the definition of read-restricted domTM given above we insisted that arbitrary domain elements could not appear on the work tape. If this restriction were dropped, then the naive simulation of a logspace read-restricted domTM by a read-restricted TM would require space $O(\log^2(n))$. It remains open whether a tighter space bound can be obtained for this simulation. Intuitively, a difficulty here is that the use of arbitary domain elements on the work tape seems to bring the ability to store information in a more compact form than is possible in a TM, since the alphabet of the domTM is arbitrarily large. On the other hand, since these objects can only be manipulated in a generic fashion, a simulation may be possible.

On a bit of a tangent, let M be an arbitrary domTM (possibly not input-order independent), let p be a polynomial, and define the mapping $f: D \to T$ by

$$f(d) = \{t \mid \text{for some listing } e \text{ of } d,\ M \text{ accepts input } et \text{ within space } p(\|et\|)\}.$$

It is straightforward to verify that f is in QPSPACE. Furthermore, it is not difficult to show that each query function in QPSPACE can be realized in this way by some domTM M and polynomial p. This provides one way of showing that the family of mappings in QPSPACE can be effectively enumerated. (Another way is provided by hypothetical datalog, which is known to have expressive power equivalent to QPSPACE [Bon88].) The question of whether there is an analogous enumeration for QPTIME remains open at this time.

Nondeterminism can be incorporated into the notion of domTMs in a natural manner, and can be used to specify *nondeterministic queries* [AV87]. In particular,

the transition function of a nondeterministic domTM maps triples to finite sets of quadruples of the correct forms. Query functions are defined by nondeterministic domTMs operating in compute mode. The requirement of input-order independence can be retained, or it can be dropped, in which case the output of the domTM computation is taken to be the union of outputs over all enumerations of the input. Using arguments similar to those of Proposition 3.3 it is easily seen that nondeterministic domTMs naturally characterize complexity classes of nondeterministic queries such as DB-NPTIME and DB-NPSPACE, as defined in [AV87, AV88b].

4. Algebras for Recursively Typed Objects

This section introduces a formal model for recursively typed complex objects, and generalizes the algebra described in the Section 2 to this model. Two results are then given. First, it is shown that without **while**, the algebra for recursively typed complex objects is $\mathscr{E}$-equivalent. On the other hand, when (nested or unnested) **while** is included, the algebra is $\mathscr{C}$-equivalent even without powerset. Hence, in this context, **while** is more powerful than powerset, contrasting to the results in the case of the conventional complex objects [AB88, GvG88].

The model we use to study recursively typed complex objects is an extension from the commonly used complex objects model and captures the essence of these objects in terms of the expressive power of their languages. Specifically, our model has a single "unrestricted" recursive type, which can be used as "leaves" in types which are otherwise non-recursive. As we shall see in the technical development, all the results and proofs can be generalized to essentially any family of types which includes at least one recursive type.

We begin by defining the "unrestricted" recursive type which contains actually all possible objects constructible using the set and tuple constructs from the atomic domain **U**.

Definition. Let **Obj** denote the smallest set such that:

(a) $\mathbf{U} \subseteq \mathbf{Obj}$;

(b) $\{X_1, ..., X_n\} \in \mathbf{Obj}$ if $0 \leqslant n$ and $X_i \in \mathbf{Obj}$ for $i \in [1...n]$; and

(c) $[X_1, ..., X_n] \in \mathbf{Obj}$ if $1 \leqslant n$ and $X_i \in \mathbf{Obj}$ for $i \in [1...n]$.

We associate the set **Obj** with an equality relationship extended from the equality of **U** in a natural manner: two atomic objects are equal if they are the same object in **U**; two tuple objects are equal if all their corresponding coordinates are equal; and two set objects are equal if every object in one set is equal to some object in the other.

The type system is now introduced in the following definition.

DEFINITION. The set of *r-types* is defined recursively as follows:

(a) U is the *atomic* r-type;

(b) *Obj* is the *universal* r-type;

(c) $\{T\}$ is a *set* r-type if T is an r-type; and

(d) $[T_1, ..., T_n]$ is a *tuple* r-type if $1 \leqslant n$ and T_i is an r-type for $i \in [1...n]$.

Two r-types are *equal* if they are the same syntactic object.

The domain of an r-type is defined similarly to that of a co-type.

DEFINITION. The *domain* of an r-type T is denoted by $\text{dom}(T)$. The domains of r-types is defined as:

(a) $\text{dom}(U) = \mathbf{U}$;

(b) $\text{dom}(Obj) = \mathbf{Obj}$;

(c) $\text{dom}(\{T\}) = \{\{X_1, ..., X_n\} \mid 0 \leqslant n \text{ and } X_i \in \text{dom}(T) \text{ for } i \in [1...n]\}$; and

(d) $\text{dom}([T_1, ..., T_n]) = \{[X_1, ..., X_n] \mid X_i \in \text{dom}(T_i) \text{ for } i \in [1...n]\}$.

Each element in $\text{dom}(T)$ is a member of **Obj**. Any finite subset of $\text{dom}(T)$ is an instance of T; $\text{inst}(T)$ denotes the family of instances of T.

It should be noted that the family of *co-types* introduced in Section 2 is a proper subset of the family of *r-types*. Recall that it is possible for two different co-types to have overlapping domains due to the presence of the set construct. In contrast, for two different r-types, it is possible that the domain of one r-type is properly contained in the domain of the other.

We now make some intuitive comments about the system of r-types. Most importantly, this system subsumes essentially all of the objects used in the calculus of Bancilhon and Khoshafian [BK89], FAD, and the Set Theoretic Data Model (STDM) of Gemstone [CM84, MSOP86]. In [BK89], special "bottom" and "top" objects are used (and FAD permits "bottom"); r-types do not provide this. Also, unlike r-typed objects, object identifiers are explicitly represented in FAD. In STDM, objects are associated with names and can be built using the set construct, which does not require elements to be of the same type. Although there is no explicit tuple construct used, it can be simulated using a set construct with object names as columns names. Also, all these models have a convention of named attributes in tuples, while here we use position to identify the coordinates.

EXAMPLE 4.1. Let $\mathbf{U} = \{a, b, c, ...\}$. The following are objects of r-type $T = [U, Obj]$,

```
[a, a], [b, [a, a]], [c, [b, [a, a]]];
```

but these are not,

```
a, [a], [a, b, c].
```

Let $\alpha = a_1 a_2 \cdots a_n$ be a finite string over **U** with $n > 0$. The following presents three naive ways to encode α into an object of r-type T, *Obj*, and $\{Obj\}$ (respectively):

$$[a_1, [a_2, ..., [a_n]]], \qquad [a_1, a_2, ..., a_n], \qquad \{a_1, \{a_2, ..., \{a_n\}\}\}.$$

The above example illustrates that the r-type system subsumes the unranked relations of QL [CH80].

The algebraic operators are extended in natural ways to range over instances of r-types. For example, we permit the formation of unions of instances of different r-types (which result in instances of r-type *Obj*). Also, horizontal operators such as selection can operate on instances of *Obj*; these "ignore" elements of the instance which do not have the right shape. Formally, we have:

DEFINITION. An *r-type assignment* τ is a (total) mapping from $\mathbf{P} \cup \mathbf{V}$ to the set of r-types.

DEFINITION. If τ is an r-type assignment and $x, y, z \in \mathbf{V}$, an *r-typed statement* is of one of the following form:

(1) $x := y$, if $\tau(x) = \tau(y)$ or $\tau(x) = Obj$;

(2)–(15) the same as in the definition of typed statement except using r-type assignment;

(16) $x := y \cup z$ (or $y \cap z$, $y - z$), if $\tau(y) \neq \tau(z)$ and $\tau(x) = Obj$;

(17) $x := \pi_{i_1, ..., i_k}(y)$, if $\tau(x) = \tau(y) = Obj$;

(18) $x := \sigma_F(y)$, if $\tau(x) = \tau(y) = Obj$.

In general, naive set theory is used as the semantics. In particular, the projection (or selection) operation in (17) (or (18)) is performed in the following way: discard all non-tuple objects and those tuple objects with insufficient width, and then do the usual projection (or selection).

DEFINITION. An *r-typed algebraic query expression* Q from a flat database schema D to a flat co-type T, denoted $Q: D \to T$, is a sequence of statements ending by an assignment to a special variable ANS of co-type T, satisfying that every predicate name referenced is in D and every variable is defined before being referenced. We denote the set of r-typed algebraic query expressions (without nested **while**, without **while**) as ALG + **while** (ALG + unnested-**while**, ALG).

Similarly, the *semantics* of an r-typed algebraic query expression $Q: D \to T$ is defined by: if d is an instance of D, the answer $Q[d]$ is the value assigned to the variable ANS, and $Q[d] = \perp$ if any **while** loop does not terminate.

When describing algebraic query expressions, we sometimes combine several operators into one non-**while** statement; put an expression in the place of the condition variable in a **while** statement; or even use calculus-like "pseudo statements," each of which is equivalent to a sequence of non-**while** statements. We present the

following example to demonstrate the algebra and to show how "big" sets can be created with while and the recursive type *Obj*.

EXAMPLE 4.2. Let x and y be variables which will hold instances of r-type *Obj* and $[\{Obj\}, Obj]$ (respectively). Initially, $x = \{a\}$ for some $a \in \mathbf{U}$. The following statements create a set of arbitrary size,

while z **do**
  $y := \text{nest}(x \times x)$;
  $x := x \cup \text{untuple}(\pi_1(y))$;
  $S_1; S_2; ...; S_n$;
end;

where the variable z is assigned value during $S_1, ..., S_n$. Note that the cardinality of x depends on the number of loops performed.

THEOREM 4.3. ALG *is* $\mathscr{E}$*-equivalent.*

Proof. Obviously, ALG has the full power of ^{co}ALG. Since ^{co}ALG is $\mathscr{E}$-equivalent by Theorem 2.5, all elementary query functions in $\mathscr{E}$ are realizable in ALG. On the other hand, for any given expression $Q: D \to T$ and a database instance d of D, since there are no loops, the number of new objects created during the process of $Q[d]$ is an elementary function of the length of Q and the number of atomic elements in d. Therefore, Q is easily seen to have hyper-exponential data complexity. By the definition of $\mathscr{E}$, Q realizes some query function in $\mathscr{E}$. ∎

An alternative proof involving a simulation of the r-typed algebra without **while** by ^{co}ALG is also possible, although it would be rather intricate.

We now turn to the algebras with **while**. The languages turn out to be very powerful. For example, we can create arbitrarily large but finite sets, as shown by Example 4.2, which can be used to store any domTM configuration. In fact, each query function in $\mathscr{C}$ can be realized in the context of unnested **while** (Theorem 4.7 and Corollary 4.8). The remainder of the section is devoted to these results. The proof technique introduced here is also used in showing similar results on deductive languages [HS93]. Since the r-types and ALG + **while** subsume the unranked relations and QL of [CH80], Theorem 4.7 could also be obtained by showing that nested **while** loops can be eliminated from the proof that QL is $\mathscr{C}$-equivalent in [CH80]. However, it is easier to ensure no nested **while**s are used in a direct proof. Also, since our model allows nested tuples and/or sets, the simulation of the domTM on all possible orderings of inputs used in this proof is simpler than that of Turing machines in [CH80].

The proof of Theorem 4.7 demonstrates that the algebra with unnested-**while** can simulate domTM computations. The essential idea is to indicate a way to store and manipulate the domTM instantaneous descriptions (IDs). Since there are many ways to encode IDs which contain symbols and strings (see Example 4.1), we first

present two examples describing the particular encoding of IDs used here, and then show that the algebras with **while** are $\mathscr{C}$-equivalent.

EXAMPLE 4.4. Let $M = (K, W, C, \delta, q_s, q_h)$ be a domTM. Without loss of generality, we assume that K and W are disjoint, $(K \cup W) \cap \mathbf{U} = \varnothing$, and $\$ \notin K \cup W \cup \mathbf{U}$. We associate to each symbol $a \in W \cup K \cup \{\$\}$ a distinct element $a_{\mathbf{U}} \in \mathbf{U}$. Suppose that 1, 2 are two elements in $\mathbf{U}$. Now let the r-type $T_{\text{sym}} = [U, U]$. We encode all symbols used by M by objects using the function $\tilde{\ }: W \cup K \cup \{\$\} \cup \mathbf{U} \to \text{dom}(T_{\text{sym}})$ ($\tilde{\ }(a)$ is denoted as $\tilde{a}$), where $\tilde{a} = [1, a]$ for each $a \in \mathbf{U}$ and $\tilde{a} = [2, a_{\mathbf{U}}]$ for each $a \in K \cup W \cup \{\$\}$.

Finally, strings representing tape contents of M are considered to be appended by an explicit end symbol "\$," and then encoded into objects of r-type $T_{\text{tape}} = [T_{\text{sym}}, Obj]$ by the function $\hat{\ }: (W \cup \mathbf{U})^* \to \text{dom}(T_{\text{tape}})$ ($\hat{\ }(a)$ is denoted as $\hat{a}$):

(1) $\hat{\varepsilon} = [\tilde{\$}, \tilde{\$}]$;

(2) $\hat{a} = [\tilde{a}, \hat{\varepsilon}]$ for $a \in W \cup \mathbf{U}$; and

(3) $\widehat{a\alpha} = [\tilde{a}, \hat{\alpha}]$ for $\alpha \neq \varepsilon$. ∎

Remark 4.5. Although the proof of Theorem 4.7 uses nested tuples to "store" the tape contents as shown in the above example, the use of nested tuples is not essential. In fact, the encoding of the domTM tape can be done in any framework which includes at least one recursive type and non-recursive set and tuple constructs. We demonstrate this by showing how to encode the domTM tape in an r-type system which uses a recursive type Obj', which has only nested set objects, in place of Obj. For encoding symbols, $T'_{\text{sym}} = \{U\}$ is used and the encoding function $\bar{a}$ is defined as: $\bar{a} = \{a\}$ if $a \in \mathbf{U}$, $\bar{a} = \{a, \varnothing\}$ otherwise. Now the tape is represented by objects of Obj' and the encoding function is: (1) $\widehat{\varepsilon} = \varnothing$; (2) $\widehat{a} = \{\bar{a}\}$; and (3) $\widehat{a\alpha} = \{\{\varnothing, \bar{a}\}, \widehat{\alpha}\}$ for $\alpha \neq \varepsilon$. Selecting the first element in the list encoded by $\widehat{\alpha}$ can then be done by selecting the element which contains the empty set. Indeed, the techniques used in this and the next section can be extended to the models of recursively typed complex objects in [BBKV87, CM84, MSOP86].

We now continue the construction of the encoding mechanism for simulating domTM in the algebra(s).

EXAMPLE 4.6. Let M be a domTM as in the previous example. Recall that an ID of M is a quintuple (q, a, α, b, β), where q is the current state, a is the content of the register, b is the symbol in the tape square where the head is, α and β are two finite strings representing the left and right portions (respectively) of the tape relative to the head. Using the r-types and the function defined in the above example, the IDs can be represented by objects of the r-type:

$$T_{\text{ID}} = [T_{\text{sym}}, T_{\text{sym}}, T_{\text{tape}}, T_{\text{sym}}, T_{\text{tape}}].$$

In particular, (q, a, α, b, β) is encoded as $[\tilde{q}, \tilde{a}, \widehat{\alpha^R}, \tilde{b}, \hat{\beta}]$.[7]

[7] α^R denotes the reverse of α.

THEOREM 4.7. ALG + unnested-**while** – powerset *is* $\mathscr{C}$*-equivalent.*

Proof. It is clear that the algebra with unnested **while** but without powerset is a procedural and generic language, and thus each query expression realizes a query function in $\mathscr{C}$. To establish that other direction, we show that every query function in $\mathscr{C}$ is realizable in ALG + unnested-**while** – *powerset.*

Let f be a C-generic query function from $D = \langle R_1 : T_1, ..., R_n : T_n \rangle$ to T in $\mathscr{C}$. From Proposition 3.3, there is an order-independent domTM $M = (K, W, C, \delta, q_s, q_h)$ which computes f. Suppose that $d = \langle R_1 : I_1, ..., R_n : I_n \rangle$ is an instance of D. In the rest of this proof, we construct a query Q and show that Q realizes f by simulating M. There are three main components in the simulation:

(1) transform the input instance d into a family of enumerations $\{e\}_{e \in \mathbf{e}}$, each of which can be used as the input for M;

(2) simulate individual steps of M (simultaneously for each enumeration in $\mathbf{e}$); and

(3) transform the output of M (if any) back to an instance of T as the result of $f(d)$.

Q "stores" the IDs of M using objects of r-type T_{ID} defined in Example 4.6 for the purpose of simulation. Essentially, Q consists of three "subqueries" corresponding to the three components described above: $Q = Q_{\mathrm{in}}; Q_{\mathrm{simu}}; Q_{\mathrm{out}}$ which have disjoint sets of variables except that the variable x^{in} of r-types T_{ID} is used by Q_{in}, Q_{simu} and the variable x^{out} of r-types T_{ID} by Q_{simu}, Q_{out}. x^{in} will be assigned value in Q_{in} and will hold the encoded initial IDs of M at the beginning of Q_{simu}. And x^{out} will be assigned value in Q_{simu} and will hold the encoded and halted IDs when Q_{out} starts.

(1) *The construction of* Q_{in}. Q_{in} will assign x^{in} the value $\{[\tilde{q}_s, \tilde{\mathcal{B}}, \hat{\varepsilon}, [, \hat{e}] \mid$ "$[e$" is an enumeration in $\mathbf{e}\}$. Note that elements in x^{in} differ only on the last column. So our focus here is to demonstrate how to generate all possible enumerations of an input instance d.

Q_{in} actually has two parts: construct for each relation R_i the family of all possible enumerations of R_i; then concatenate all those families.

Now let x be a variable of r-type $[\{T_i\}, T_{\mathrm{tape}}]$ (note that x will hold an instance of the r-type). Initially, x has the value $\{[I_i, \widehat{\ }\,]\}$. After execution of the following loop, x_i^{in} of r-type T_{tape} will have the set of all encoded enumerations,

while collapse · untuple · $\pi_1(x)$ **do**
  $y := \{[r, \hat{e}, t, t'] \mid [r, \hat{e}] \in x; t, t' \in r; t \neq t'\}$;
  $z := \{[r - \{t\}, \hat{e}, t] \mid [r, \hat{e}, t, t'] \in y\}$;
  $x := \{[r, \widehat{te}] \mid [r, \hat{e}, t] \in z\}$;
end;
$x_i^{\mathrm{in}} := \{\widehat{[e} \mid [\varnothing, \hat{e}] \in x\}$;

where y can be constructed from x by duplicating the first coordinate twice (two cross products and one selection with the condition $(1=3 \wedge 1=4)$), unnesting the third and fourth columns, followed by a selection with $(3 \neq 4)$; z can be obtained from y by nesting the fourth column and then projecting on the fourth, second, and third columns.

Having all x_i^{in} $(i \in [1 \ldots n])$ computed, the concatenation is essentially a series of merges following cross products. Suppose $[\hat{e}_1, \hat{e}_2]$ is a tuple of two encoded enumerations. To merge them into $\widehat{e_1 e_2}$ we first invert $\hat{e}_1$ using a **while** to obtain $[\widehat{e_1^{\text{R}}}, \hat{e}_2]$ and then repeatedly insert the top element of e_1^{R} on to the top of $\hat{e}_2$ by another **while** loop. Let x be the variable holding the enumerations after concatenations. We assign x^{in}:

$$x^{\text{in}} := \{[\tilde{q}_s, \tilde{\not{b}}, \hat{e}, \tilde{[}, \hat{e}] \mid \text{“}[e\text{”} \in x\}.$$

Hence, Q_{in} generates the set of inputs for M. Note that Q_{in} uses only $2n-1$ unnested **while** statements.

(2) *The construction of* Q_{simu}. The input for Q_{simu} is the variable x^{in} holding a set of encoded initial IDs for M. Q_{simu} is required to simulate the behaviors of M on these inputs and assign a set of outputs of M (in the format of IDs, i.e., objects of r-type T_{ID}) to x^{out} if M halts on any of the inputs. Q_{simu} is accomplished by a “big” **while** loop, namely

$x := x^{\text{in}}$;
while $\{\tilde{q}_h\} - (\text{untuple} \cdot \pi_1(x))$ **do**
  $Q_{\delta(q,a,b)}$; /* for each transition $\delta(q, a, b)$ */
  $x := \bigcup \{z^{\text{out}}_{\delta(q,a,b)} \mid \delta(q, a, b)$ is a transition$\}$;
end;
$x^{\text{out}} := \{[\tilde{q}_h, y_2, y_3, y_4, y_5] \in x\}$;

where $Q_{\delta(q,a,b)}$ is a sequence of statements (described shortly) which performs one step of the simulation(s) according to the transition $\delta(q, a, b)$ and stores the resulting ID(s) in the variable $z^{\text{out}}_{\delta(q,a,b)}$ (of r-type T_{ID}). When at least one simulation halts, x^{out} simply selects those halted IDs.

We describe how $Q_{\delta(q,a,b)}$ can be constructed with the following two examples:

- Suppose that $\delta(\text{q}, \text{a}, \text{b}) = (\text{q}_1, \text{a}_1, \text{b}_1, \text{R})$ and $\text{a}, \text{b} \notin \{\eta, \kappa\}$. Then, $Q_{\delta(\text{q},\text{a},\text{b})}$ is as follows:

$$z^{\text{in}}_{\delta(\text{q},\text{a},\text{b})} := \{[\tilde{\text{q}}, \tilde{\text{a}}, \hat{e}_L, \tilde{\text{b}}, \hat{e}_{\text{R}}] \in x\};$$

$$x := x - z^{\text{in}}_{\delta(\text{q},\text{a},\text{b})};$$

$$z^{\text{out}}_{\delta(\text{q},\text{a},\text{b})} := \{[\tilde{\text{q}}_1, \tilde{\text{a}}_1, \widehat{\text{b}_1 e_L}, \tilde{\text{c}}, \hat{e}'_R] \mid [\tilde{\text{q}}, \tilde{\text{a}}, \hat{e}_L, \tilde{\text{b}}, \hat{e}_R] \in z^{\text{in}}_{\delta(\text{q},\text{a},\text{b})} \wedge e_R = \text{c}e'_R \neq \varepsilon\}$$
$$\cup \{[\tilde{\text{q}}_1, \tilde{\text{a}}_1, \widehat{\text{b}_1 e_L}, \tilde{\not{\text{b}}}, \hat{\varepsilon}] \mid [\tilde{\text{q}}, \tilde{\text{a}}, \hat{e}_L, \tilde{\text{b}}, \hat{\varepsilon}] \in z^{\text{in}}_{\delta(\text{q},\text{a},\text{b})}\}.$$

• Suppose that $\delta(q, \eta, \eta) = (q_1, \eta, a, —)$. Let x_C be a variable holding the value $\{\tilde{c} \mid c \in C\}$, where C is the set of constants used by M. $Q_{\delta(q, \eta, \eta)}$ is then expressed as

$$z^{\text{in}}_{\delta(q, \eta, \eta)} := \{[\tilde{q}, y, \hat{e}_L, y', \hat{e}_R] \in x \mid y = y' \wedge y \notin x_C \wedge y' \notin x_C\};$$
$$x := x - z^{\text{in}}_{\delta(q, \eta, \eta)};$$
$$z^{\text{out}}_{\delta(q, \eta, \eta)} := \{[\tilde{q}_1, y, \hat{e}_L, \tilde{a}, \hat{e}_R] \mid [\tilde{q}, y, \hat{e}_L, y, \hat{e}_R] \in z^{\text{in}}_{\delta(q, \eta, \eta)}\};$$

If the transition is $\delta(q, \eta, \kappa)$, then in the step selecting IDs we only change the condition $y = y'$ to $y \neq y'$.

In general, every $Q_{\delta(q, a, b)}$ has the similar structure of three "pseudo statements": the first selects applicable IDs into the variable $z^{\text{in}}_{\delta(q, a, b)}$; the second removes these from further consideration in the same round; and the third modifies the IDs and put them into $z^{\text{out}}_{\delta(q, a, b)}$.

(3) *The construction of* Q_{out}. Q_{out} is an inverse procedure of Q_{in}. Using a similar technique, this can be expressed using one unnested **while**. In the decoding it is important that a variable of co-type $T = [U, ..., U]$ be used to gather the tuples of encoded outputs of M. This will ensure that the output of Q is of flat co-type T rather than r-type $[Obj, ..., Obj]$. Since M is order independent, this will yield the desired result. Hence, Q realizes the query function f in $\mathscr{C}$. ∎

COROLLARY 4.8. ALG + while − *powerset and* ALG + **while** *are also* $\mathscr{C}$*-equivalent.*

We conclude this section by the following remark. The "magic power" of the recursively typed objects is their ability to represent arbitrarily large objects, in the forms of either nested tuples or sets of arbitrary nesting levels, arbitrarily wide tuples, or arbitrarily large but finite sets, all built without using "invented values" (relative to indom(d)). Together with appropriate control structures (e.g., **while**), it then yields rich expressiveness.

5. THE CALCULUS AND THE ARITHMETIC HIERARCHY

In this section, we characterize the expressive power of the calculus for recursively typed complex objects and establish the correspondence between recursively typed objects and invented values. In particular, we show that in this context the expressive power of the calculus is that of the arithmetical hierarchy (restricted to query functions), or equivalently, query functions computed by Turing machines with (finitely nested) oracles. We also show that the r-typed calculus has expressive power equivalent to the previously introduced co-typed calculus with countable invention [HS88]. Finally, we compare the r-typed calculus and the co-typed calculus with finite invention.

The section begins by introducing the r-typed calculus and the class $\mathscr{A}$ of arithmetical query functions and stating the main theorem. As the first step in proving the theorem, the co-typed calculus with invention is introduced and shown equivalent to the r-typed calculus. After this the theorem is demonstrated. It should be noted that the results presented here are true for any r-type system which contains at least one recursive type, in an argument similar to Remark 4.5.

Consider now the issue of extending ^{co}CALC to recursively typed complex objects while retaining the general spirit of typing variables. Recall that for two different r-types T_1, T_2, their domains may overlap. Hence, it is legitimate to write $x = y$ even if x and y are of different r-types. Second, the term $x.i$ should be allowed when x if of r-type *Obj*. However, the domain of r-type *Obj* may contain non-tuple objects as well. A new predicate is needed to resolve this problem since in ^{co}CALC it is impossible to determine syntactically if an object is a tuple. Following [KV84], we prohibit terms $t.i$ and introduce predicates π_i, where $x\pi_i y$ tests whether y is a tuple and has ith column equal to x. Similarly, the predicate $\in$ in $x \in y$ will first test if y is a set.

A *term* is either a variable x, or a constant $a \in \mathbf{U}$. If s, t are terms and $P \in \mathbf{P}$ is a predicate name, then an *atomic formula* is either (1) $s = t$, or (2) $s \in t$, or (3) $P(s)$, or (4) $s\pi_i t$ for $i > 0$. A *formula* is either an atomic formula; or built from other formulas using $\neg$, $\wedge$, $\vee$; or built using quantifications $\forall x/T\phi$, $\exists x/T\phi$, where T is an r-type and ϕ is a formula, where the relaxed typing of the variables is respected.

DEFINITION. Let D be a flat database schema and T a flat co-type. An *r-typed calculus query expression* Q from D to T, denoted $Q: D \to T$, is an expression of form: $Q = \{t/T \mid \phi(t)\}$, where $t \in \mathbf{V}$ and $\phi(t)$ is a formula having only free variable t, such that all predicate names referenced in ϕ are in D. Let CALC denote the family of all r-typed calculus query expressions.

Given a database instance d, the *semantics* of a query $Q = \{t/T \mid \phi(t)\}$, denoted $Q[d]$, is defined naturally as: $\{o \mid \text{adom}(o) \subseteq \text{adom}(d, Q) \wedge \phi(o) \text{ is true}\}$ using the limited interpretation. That is, the formula $\exists x/S\phi(x)$ is true if there is an object $o \in \text{dom}(S)$ with $\text{adom}(o) \subseteq \text{adom}(d, Q)$ such that $\phi(o)$ is true; the formula $\forall x/S\phi(x)$ is true if for each object $o \in \text{dom}(S)$ with $\text{adom}(o) \subseteq \text{adom}(d, Q)$, $\phi(o)$ is true. Note that if *Obj* occurs in the r-type S, then these quantifiers range over an infinite set of objects. Since objects in the answer of $Q[d]$ are "flat" tuples built from elements in $\text{adom}(d, Q)$, $Q[d]$ is always finite and thus defined.

We now introduce the arithmetical hierarchy. Following [Rog87] we have:

DEFINITION. Let Σ be an alphabet. A (possibly infinite) predicate P of arity $n > 0$ is in the *arithmetical hierarchy* if it is the result of a sequence of projection and complementation operations on a recursive predicate.

It is well known [Rog87] that a predicate P is in the arithmetical hierarchy iff it can be expressed as $\{(y_1, ..., y_n) \mid Q_1 x_1 \cdots Q_m x_m R(x_1, ..., x_m, y_1, ..., y_n)\}$, where R is a recursive relation and Q_i is a quantifier for each $i \in [1...m]$.

Let $L \subseteq \Sigma^*$. As defined in [HU79], a *Turing machine with oracle L*, denoted M^L, is a single-tape Turing machine with three special states $q_?$, q_y, q_n, which are used to test membership in L. Now let $\mathbf{O}_0 = \{M \mid M \text{ is a Turing machine}\}$; and for $i \geqslant 1$, $\mathbf{O}_i = \{M^L \mid L \text{ is a language accepted by some Turing machine or by } M_1^{L_1} \in \mathbf{O}_j \text{ for } j < i\}$. And finally, define $\mathbf{O} = \bigcup_{i \geqslant 0} \mathbf{O}_i$.

For a query function f, the *graph* of f is $\mathrm{Gr}(f) = \{(d, t) \mid t \in f(d)\}$. Using the well-known equivalence between the arithmetical hierarchy and languages computable by Turing machines with finitely nested oracles it is straightforward to verify that:

PROPOSITION 5.1. *A query function f is computable by a Turing machine in* **O** *iff* $\mathrm{Gr}(f)$ (*under a suitable encoding*) *is in the arithmetical hierarchy.*

With this background we now have:

DEFINITION. The family $\mathscr{A}$ of *query functions in the arithmetical hierarchy* is the set of total query functions f such that f is computable by some Turing machine with oracle $M^A \in \mathbf{O}$.

Note that every query function in $\mathscr{A}$ is total. Since partial functions can be obtained by having one more level of oracles, this does not reduce the expressiveness. It is clear that natural analogs of the arithmetical hierarchy and Turing machines with oracles could be made using domTMs and "alphabets" which include **U**. This would permit a more direct definition of the class $\mathscr{A}$.

THEOREM 5.2. CALC *is* $\mathscr{A}$*-equivalent.*

To prove this theorem we first introduce the co-typed calculus with countable invention and show it to be equivalent to CALC. For expediency we also introduce the notion of finite invention at this time. (More complete descriptions of these languages are found in [HS88].) For a query $Q \in {}^{\mathrm{co}}\mathrm{CALC}$, a database d, and $i \leqslant \omega$, the semantics of Q under d with i invented values, denoted $Q|_i[d]$, is obtained as follows: (a) evaluate Q under limited interpretation with the active domain extended to include i new values, denoting the result $Q|^i[d]$; and (b) delete from $Q|^i[d]$ objects containing invented values. Note that the limited interpretation of Q is $Q|_0[d]$, and (assuming a countably infinite universal domain) the unlimited interpretation of Q is $Q|^\omega[d]$.

DEFINITION. If $Q \in {}^{\mathrm{co}}\mathrm{CALC}$, the semantics of Q under *finite invention*, Q^{fi}, is defined as $Q^{\mathrm{fi}}[d] = \bigcup_{0 \leqslant i < \omega} Q|_i[d]$ for all database instances d; the semantics of Q under *countable invention*, Q^{ci}, is defined as $Q^{\mathrm{ci}}[d] = Q|_\omega[d]$ for all d. Let ${}^{\mathrm{co}}\mathrm{CALC}^{\mathrm{fi}}$ and ${}^{\mathrm{co}}\mathrm{CALC}^{\mathrm{ci}}$ denote the families of calculus queries with finite and countable invention semantics (respectively).

To illustrate the power of ${}^{\mathrm{co}}\mathrm{CALC}^{\mathrm{ci}}$ and ${}^{\mathrm{co}}\mathrm{CALC}^{\mathrm{fi}}$ we include:

EXAMPLE 5.3 [HS88]. Let M be a (conventional) Turing machine with unary input alphabet $\{a\}$; and let c be a constant in **U**. Then there is a query Q in ^{co}CALC^{fi} which computes the total function

$$f_{\text{halt}}(d) = \begin{cases} \{[c]\} & \text{if } M \text{ halts on } a^{|d|}; \\ \varnothing & \text{otherwise} \end{cases}$$

and there is a query Q' in ^{co}CALC^{ci} which computes the query

$$f_{\overline{\text{halt}}}(d) = \{[c]\} - f_{\text{halt}}(d).$$

Intuitively, the body of Q outputs the tuple $[c]$ if there exists a halting computation of M on the input $a^{|d|}$ whose running times is $\leqslant$ the number of active domain and invented objects. Because the semantics of Q is obtained by taking the union of its output on all finite sets of invented values, Q essentially has access to computations of M of all possible lengths. Now we turn to $f_{\overline{\text{halt}}}$. It is first observed that the graph of $f_{\overline{\text{halt}}}$ is not recursively enumerable while the answer to any query in ^{co}CALC^{fi} is. It follows that $f_{\overline{\text{halt}}}$ is not expressible within ^{co}CALC^{fi} [HS88]. On the other hand, under countable invention, there is a query Q' which can state "there are *no* halting computations of M on the input," or can simultaneously examine all possible computations of M on the input. Hence Q' expresses $f_{\overline{\text{halt}}}$. ∎

We conclude our review of ^{co}CALC^{fi} and ^{co}CALC^{ci} with:

PROPOSITION 5.4 [HS88]. (1) ^{co}CALC^{ci} is strictly more powerful than ^{co}CALC^{fi}, which in turn is strictly more powerful than $\mathscr{C}$; (2) f is realizable in ^{co}CALC^{fi} if and only if $\text{Gr}(f)$ is recursively enumerable.

There is an intuitive correspondence between recursively typed objects and invented values in this context. Consider a query Q whose formula contains a quantified variable x of the r-type $\{Obj\}$. Then x ranges over arbitrarily large finite sets, all constructed from atomic objects appearing either in the input database instance or as constants in Q (if any). The elements in x can be used in the same manner as invented values. This leads to the following:

THEOREM 5.5. CALC $\equiv$ ^{co}CALC^{ci}.

Proof. To show that ^{co}CALC$^{ci} \sqsubseteq$ CALC, note that if a is a constant then the set $\text{cons}_{Obj}(\{a\})$ is countably infinite; this can be used as the set of invented values. For the opposite direction, let Q be a query in CALC. The central problem in building a query Q' in ^{co}CALC^{ci} which simulates Q is the removal of the r-type Obj wherever it occurs in Q. This is accomplished by "flattening" each element of $\text{cons}_{Obj}(\text{adom}(d, Q))$ into an object of co-type $\{[U, U, U, U]\}$ which uses invented values. This flattening is reminiscent of the representation of complex objects used in the logical data model [KV84]. (The proof of Lemma 6.5 in [HS88] uses this

technique of flattening in a slightly different context and illustrates how it can be simulated in ^{co}CALC.) ▮

We are now prepared to present:

Proof of Theorem 5.2. To see that every query expression in CALC $\equiv$ ^{co}CALC^{ci} is in $\mathscr{A}$, let $Q = \{t/T \mid \phi(t)\}$ be in ^{co}CALC^{ci}. Without loss of generality, we assume that $\phi(t) = \mathscr{Q}\bar{x}_i \cdots \mathscr{Q}\bar{x}_n \psi(t, \bar{x}_1, ..., \bar{x}_n)$ is in prenex normal form, where each $\mathscr{Q}\bar{x}_i$ is a block of quantified variables with the same quantifier, and alternating with each block. It is easily seen that there is some Turing machine M which accepts (an encoding of) $\{(t, \bar{x}_1, ..., \bar{x}_n) \mid \psi(t, \bar{x}_1, ..., \bar{x}_n)\}$. Then, working backwards inductively it is straightforward to build, for each $i \in [1...n]$, a Turing machine with oracle which accepts the encoding of the set $\{(t, \bar{x}_1, ..., \bar{x}_i) \mid \mathscr{Q}\bar{x}_{i+1} \cdots \mathscr{Q}\bar{x}_n \psi(t, \bar{x}_1, ..., \bar{x}_n)\}$. It follows that the query function defined by Q is in $\mathscr{A}$.

For the other direction, let f be a query function in $\mathscr{A}$. By Proposition 5.1, Gr(f) is in the arithmetical hierarchy. This implies that under a suitable encoding $\hat{}$ the set $\{(\hat{d}, \hat{t}) \mid t \in f(d)\} = \{(y_1, y_2) \mid Q_1 x_1 \cdots Q_n x_n R(x_1, ..., x_n, y_1, y_2)\}$, where R is a recursive relation. Using binary relations to simulate strings it is now straightforward to simulate $Q_1 x_1 \cdots Q_n x_n R(x_1, ..., x_n, y_1, y_2)\}$ using ^{co}CALC^{ci}. This yields the result. ▮

Consider a sublanguage CALC$_\exists$, the class of calculus queries whose variables of recursive r-types are all existentially quantified at the beginning of the query formula. (A *recursive r-type* is an r-type in which *Obj* occurs.) it is interesting to compare this with ^{co}CALC^{fi}. Using an argument similar to the proof of the Theorem 5.5 based on encoding of arbitrary objects, it can be shown that every query in CALC$_\exists$ can be simulated by some query in ^{co}CALC^{fi}. At first glance, since CALC$_\exists$ has only one alternation of quantifiers and variables whose r-types do not involve *Obj* range over finite sets, it would appear that CALC$_\exists$ is equivalent to the first layer of the arithmetic hierarchy (denoted as Σ_1 in [Rog87]). Furthermore, it would appear that a query Q, which has the form $\{t/T \mid \exists x/\{Obj\}\ \phi(t, x)\}$, where $\phi(t, x)$ does not use any universal quantifiers on variables of r-type involving *Obj*, can be constructed to simulate some Turing machine M. However, this argument fails because a universal quantification over elements of x is necessary in this naive simulation and the r-type of those elements will be *Obj*. It remains open whether CALC$_\exists \equiv$ ^{co}CALC^{fi}.

It should be noted that the results on expressive power of ^{co}CALC with invented values rely on the presence of variables ranging over set co-types. For example, it was demonstrated in [HS88] (details appear in [HS89a]) that the relational calculus with (finite or countable) invention has only the expressive power of the conventional relational calculus. In other words, the relational calculus with the unlimited interpretation has the same expressive power as with the limited interpretation. One of the two key ideas of that result was shown in an independent investigation of [AGSS86].

6. Conclusion

The results described in this paper characterize the expressive power of algebraic and calculus query languages based on conventional paradigms from databases, but interpreted in a context where recursively typed complex objects are permitted. The languages studied cover a spectrum of expressive powers, ranging from the algebra without **while**, which is equivalent to $\mathscr{E}$; through the algebra with **while**, which has the power of $\mathscr{C}$; up to the calculus, which has the power of $\mathscr{A}$.

The companion paper [HS93] studies analogous questions for deductive query languages. As noted in the Introduction, several deductive languages supporting complex objects have been proposed, including COL [AG88], the recursive language of [AB88], LDL [NT89], and the calculus of Bancilhon and Khoshafian (called here the BK-calculus) [BK89]. The first languages permit only non-recursive types and are relatively narrow in their focus. Two extensions of COL to incorporate recursive types, based on the stratified and inflationary semantics (respectively), are studied in [HS93]. It is shown that both cases yield the expressive power of the computable queries. It is also shown in [HS93] that in the case of non-recursive types, COL under inflationary semantics has the same expressive power as under stratified semantics, namely that of $\mathscr{E}$. Analogous results hold for the recursive language of [AB88]. This provides an interesting contrast to the fact that in the relational model stratified DATALOG$^\neg$ is strictly weaker than inflationary DATALOG$^\neg$ [Kol87, AV88a].

LDL supports recursive types, freely interpreted function symbols (and hence lists), and some arithmetic operations. The presence of lists in LDL can be used to show that it is $\mathscr{C}$-equivalent, using arguments essentially the same as those used for COL with recursive types. Curiously, as shown in [HS93], LDL without freely interpreted function symbols or arithmetic operations is only $\mathscr{E}$-equivalent. This results from the form of stratification imposed on set-formation in LDL, which is more restrictive than that found in COL.

The deductive language BK-calculus, which supports recursive types, is distinguished by its use of the "sub-object" relationship instead of "$=$" in defining the semantics of rule application. As a result, the techniques appropriate for studying its expressive power are significantly different than for the other deductive languages. As shown in [HS93], when provided with suitably encoded input, the BK-calculus can simulate arbitrary Turing machines. On the other hand, when restricted to relational input and output, the expressive power of the BK-calculus is equivalent to a natural variant of the conjunctive queries of [CM77]. This variant is strictly weaker than the conjunctive queries; for example, it cannot compute the natural join.

APPENDIX: Proof of Proposition 3.5

This appendix presents a proof of Proposition 3.5, which is re-stated here for the reader's convenience.

PROPOSITION 3.5. *The family of query functions tested by read-restricted* domTMs *operating in logspace is* QLOGSPACE.

Suppose that M' is a TM with separate read and work tapes that operates in space $O(\log n)$ and tests the C-generic query function $f: D \to T$ using encoding μ. We sketch the construction of a read-restricted domTM M which simulates M' and also operates in space $O(\log n)$. Suppose that M is given as input the word et, where e is an enumeration of some input instance d. Let the encoding v be defined so that $v(c) = \mu(c)$ for each $c \in C$, and let $v(a) = \alpha$, where α is the ith element in the lexicographic ordering of $\{0, 1\}^* - \mu[C]$ if a is the ith distinct member of $\text{adom}(d) - C$ occurring in the enumeration e (reading from left to right). M will simulate the behavior of M' on input $v(e)$. By an argument analogous to that of Lemma 3.2, this will ensure that M returns the correct answer.

In the simulation of M', M will use eight separate areas of its work tape, which we denote as $T1$ through $T8$. At the end of a simulated move of M',

$T1$ holds the actual contents of the work tape of M';

$T2$ holds the position (encoded as a binary number using $\{0, 1\}^*$) of the work tape head of M in the word stored in $T1$;

$T3$ holds information about the location of the read tape head of M'; and

$T4$ holds $v(a)$, if the read head of M is on a square containing $a \in U$, and is empty otherwise.

To clarify the role of $T3$ and $T4$, suppose that $v(a)$ has length k and that, at some point in its computation, M' is reading the ith letter of some occurrence of $v(a)$. Then the read head of M will be positioned at the corresponding occurrence of a, $T3$ will hold the number i (encoded in $\{0, 1\}^*$), and $T4$ will hold $v(a)$.

We now sketch how $v(a)$ is determined for $a \in \text{adom}(d)$. Suppose that the read head of M' has just moved onto the encoding $v(a)$ of some atomic object a. In M, the read tape head will move onto an occurrence of a. If $a \in C$, then $v(a)$ is immediately generated by M. Otherwise, M places a into its register and then moves is read head from that occurrence of a to the left end of the work tape, counting the squares in $T5$ so that it can remember where that occurrence is. M now sweeps right to find the first occurrence of a on the work tape and records that location in $T6$. Now M will use $T7$ and $T8$ to count the number of distinct elements of $\mathbf{U} - C$ occurring between that first a and the left end of the tape. M will start at the left end of the tape and use $T7$ as a counter to remember the current tape square being considered. Suppose that $b \in C$ is in that tape square. Then M ignores it and moves right, incrementing $T7$. If $b \notin C$, then M will place b in its register and sweep left to see if any b's occurred previously. If so, the counter in $T8$ is not incremented, and if not, $T8$ is incremented. $T7$ is used to return to the square immediately right of the occurrence of b just considered. When $T7$ has the same number as $T6$, the first occurrence of a has been reached. $T8$ now holds the number i of distinct elements of $\text{adom}(d) - C$ occurring in e before the first occurrence of a. This number is now changed into the $(i+1)$th element $\alpha = v(a)$ of the

lexicographic ordering of $\{0, 1\}^* - \mu[C]$ and placed into $T4$. Finally, $T3$ is given the number 1 or $\|v(a)\|$, depending on whether M entered the occurrence of a from the right or the left.

During a simulated move of M', the current contents of square number $T2$ of $T1$ is obtained by sweeping across $T1$, replacing each symbol x by $\bar{x}$, and decrementing a copy of $T2$ in the process.

This completes the sketch of how M simulates M'. We conclude by considering the space used by M in this simulation. In particular, we provide bounds on the space used by each part of the work tape:

$T1$: space used by $M' \leqslant \log \|v(e)\| \in O(\log(n \log n)) \leqslant O(\log n)$.

$T2$: $\leqslant \log(\text{space } M' \text{ uses}) \in O(\log \log n) \leqslant O(\log n)$.

$T3$: $\leqslant O(\log \log n) \leqslant O(\log n)$

$T4$: $\leqslant O(\log n)$

$T4$ through $T7$: $\leqslant \log n$

$T8$: $\leqslant \log \log n$

Thus, M uses space $O(\log n)$ on inputs of length n.

Suppose now that M is a read-only domTM that tests the C-generic query function f in space $O(\log n)$. We describe a TM M with separate read and work tapes which simulates M. Let M' have input $\mu(e)\,\mu(t)$ for some encoding μ, enumeration e of input instance d, and tuple t. Note that each encoded atomic object $\mu(a)$ corresponding an atomic object a occurring in e is delimited by punctuation symbols in $\mu(e)$. In the simulation of M, M' will treat an encoded atomic object as a unit. It can easily determining for an encoded atomic object $\mu(a)$ whether $a \in C$ or not.

M' will have three sections of its work tape, used as follows:

$S1$ holds the (encoded) contents of the work tape of simulated computation of M;

$S2$ holds a counter which indicates, at a given step of the simulation, the position of M's tape head in $S1$; and

$S3$ holds the symbol x (encoding $\mu(a)$) when M's register is holding a symbol x (atomic object a).

It is straightforward to verify that M' will simulate M correctly and will operate within space $O(\log n)$. ∎

ACKNOWLEDGMENT

The authors thank Dino Karabeg for stimulating discussions concerning domTMs. The authors also thank Serge Abiteboul and anonymous referees for their detailed comments leading to the improvements of the paper.

REFERENCES

[AB88] S. ABITEBOUL AND C. BEERI, "On the Power of Languages for the Manipulation of Complex Objects," Technical Report No. 846, INRIA, May 1988.

[AG91] S. ABITEBOUL AND S. GRUMBACH, A rule-based language with functions and sets. *ACM Trans. on Database Systems* **16**, No. 1 (1991), 1–30.

[AGSS86] A. K. AYLAMAZYAN, M. M. GILULA, A. P. STOLBOUSHKIN, AND G. F. SCHWARTZ, Reduction of the relational model with infinite domain to the case of finite domains [Russian], *Proc. USSR Acad. Sci. (Dokl.)* **286**, No. 2 (1986), 308–311 (translated by Dina O. Goldin, Brown University, October, 1986).

[AV87] S. ABITEBOUL AND V. VIANU, "Transaction Languages for Database Update and Specification," Technical Report No. 715, INRIA, September 1987.

[AV88] S. ABITEBOUL AND V. VIANU, "Datalog Extensions for Data base Queries and Updates," Technical Report 900, INRIA, September 1988; *J. Comput. System Sci.*, **43**, No. 1 (1991), 62–124.

[AV90] S. ABITEBOUL AND V. VIANU, Procedural languages for database queries and updates, *J. Comp. System Sci.* **41** (1990), 181–229.

[AV91] S. ABITEBOUL AND V. VIANU, Generic computation and its complexity, *in* "Proc. ACM SIGACT Symp. on the Theory of Computation, 1991," pp. 209–219.

[BBKV87] F. BANCILHON, T. BRIGGS, S. KHOSHAFIAN, AND P. VALDURIEZ, FAD, a powerful and simple database language, *in* "Proc. Int. Conf. on Very Large Data Bases, 1987," pp. 97–105.

[BK89] F. BANCILHON AND S. KHOSHAFIAN, A calculus for complex objects, *J. Comput. System Sci.* **38**, No. 2 (1989), 326–340.

[Bon88] A. J. BONNER, Hypothetical datalog: Complexity and expressibility, *in* "Proc. 2nd Intl. Conf. on Database Theory," Lecture Notes in Computer Science, Vol. 326, pp. 144–160, Springer-Verlag, New York/Berlin, 1988.

[CH80] A. K. CHANDRA AND D. HAREL, Computable queries for relational data bases, *J. Comput. System Sci.* **21**, No. 2 (1980), 156–178.

[CH82] A. K. CHANDRA AND D. HAREL, Structure and complexity of relational queries, *J. Comput. System Sci.* **25**, No. 1 (1982), 99–128.

[CM77] A. K. CHANDRA AND P. M. MERLIN, Optimal implementation of conjunctive queries in relational data bases, *in* "Proc. ACM SIGACT Symp. on the Theory of Computing, 1977," pp. 77–90.

[CM84] G. COPELAND AND D. MAIER, Making Smalltalk a database system, *in* "Proc. ACM SIGMOD Int. Conf. on the Management of Data, 1984."

[Cod70] E. F. CODD, A relational model of data for large shared data banks, *Comm. ACM* **13**, No. 6 (1970), 377–387.

[GvG88] M. GYSSENS AND D. VAN GUCHT, The powerset algebra as a result of adding programming constructs to the nested relational algebra, *in* "Proc. ACM SIGMOD Int. Conf. on Management of Data, 1988."

[HK87] R. HULL AND R. KING, Semantic data modeling: Survey, applications, and research issues, *ACM Comput. Surveys* **19**, No. 3 (1987), 201–260.

[HS88] R. HULL AND J. SU, "On the Expressive Power of Database Queries with Intermediate Types," Technical Report 88-53, Computer Science Department, University of Southern California, 1988; *J. Comput. System Sci.* **43**, No. 1 (1991), 219–267.

[HS89a] R. HULL AND J. SU, "Domain Independence and the Relational Calculus, Technical Report 88-64, Computer Science Dept., University of Southern California, 1989; revised 1991.

[HS89b] R. HULL AND J. SU, "Untyped Sets in Database Query Languages," Technical Report 89-08, Dept. of Computer Science, University of Southern California, September 1989.

[HS93] R. HULL AND J. SU, Deductive query languages for recursively typed complex objects, 1993, in preparation.

[HU79] J. E. HOPCROFT AND J. D. ULLMAN, "Introduction to Automata Theory, Languages, and Computation," Addison–Wesley, Reading, MA, 1979.

[Hl87] R. HULL, A survey of theoretical research on typed complex database objects, *in* "Databases" (J. Paredaens, Ed.), pp. 193–256, Academic Press, London, 1987.

[Jac82] B. JACOBS, On database logic, *J. Assoc. Comput. Mach.* **29**, Vol. 2 (1982), 310–332.

[Kol87] P. G. KOLAITIS, The expressive power of stratified logic programs, manuscript, Stanford University, 1987, to appear in *Information and Computation*, 1991.

[Kup87] G. M. KUPER, Logic programming with sets, *in* "Proc. ACM Symp. on Principles of Database Systems, pp. 1987," pp. 11–20.

[KV84] G. M. KUPER AND M. Y. VARDI, A new approach to database logic, *in* "Proc. ACM Symp. on Principles of Database Systems, 1984," pp. 86–96.

[KV88] G. M. KUPER AND M. Y. VARDI, On the complexity of queries in the logical data model, *in* "ICDT'88—Proc. 2nd Int. Conf. on Database Theory" (M. Gyssens, J. Paredaens, and D. van Gucht, Eds.), Lecture Notes in Computer Science, Vol. 326, pp. 267–280, Springer-Verlag, New York/Berlin, 1988.

[MSOP86] D. MAIER, J. STEIN, A. OTIS, AND A. PURDY, Development of an object-oriented DBMS, *in* "Proc. Conf. on OOPSLA, 1986," pp. 472–482.

[NT89] S. NAQVI AND S. TSUR, "A Logical Language for Data and Knowledge Bases," Computer Sci., New York, 1989.

[PvG88] J. PAREDAENS AND D. VAN GUCHT, Possibilities and limitations of using flat operators in nested algebra expressions, *in* "Proc. ACM Symp. on Principles of Database Systems, Austin, Texas, 1988."

[RKS88] M. A. ROTH, H. F. KORTH, AND A. SILBERSCHATZ, Extended algebra and calculus for nested relational databases, *ACM Trans. Database Systems* **13**, No. 4 (1988), 389–417.

[Rog87] H. ROGERS, JR. "Theory of Recursive Functions and Effective Computability," MIT Press, Cambridge, MA, 1987.

[She90] Y.-H. SHENG, Idlog: Extending the expressive power of deductive database languages, *in* "Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990," pp. 54–63.

[Var82] M. Y. VARDI, The complexity of relational query languages, *in* "Proc. ACM SIGACT Symp. on the Theory of Computing, 1982," pp. 137–146.

[Var83] M. Y. VARDI, Review of [Jac82], "Zentralblatt für Mathematik, 497.68061, 1983."