Journal of Computer and System Sciences, 43(1):219-267, August, 1991

On the Expressive Power of Database Queries with Intermediate Types*

Richard Hull and Jianwen Su†

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

Received January 8, 1989; revised January 22 1990

The *set-height* of a complex object type is defined to be its level of nesting of the set construct. In a query of the complex object calculus which maps a database D to an output type T, an *intermediate type* is a type which is used by some variable of the query, but which is not present in D or T. For each $k, i \geq 0$ we define $\text{CALC}_{k,i}$ to be the family of calculus queries mapping from and to types with set-height $\leq k$ and using intermediate types with set-height $\leq i$. In particular, $\text{CALC}_{0,0}$ is the classical relational calculus, and $\text{CALC}_{0,1}$ is equivalent to the family of second-order (relational) queries. Several results concerning these families of languages are obtained. A primary focus is on the families $\text{CALC}_{0,i}$, which map relations to relations. Upper and lower bounds in terms of hyper-exponential time and space on the complexity of these families are provided. The $\text{CALC}_{0,i}$ hierarchy does not collapse with respect to expressive power. The union $\cup_{0 \leq i} \text{CALC}_{0,i}$ is exactly the family of *elementary queries*, i.e., queries with hyper-exponential complexity. The expressive power of queries from the complex object calculus interpreted using semantics based on the use of arbitrarily large finite or infinite set of *invented values* is studied. Under these semantics, the expressive power of the relational calculus is not increased, and the $\text{CALC}_{0,i}$ hierarchy collapses at $\text{CALC}_{0,1}$. In general, queries with these semantics may not be computable. We also consider an alternative semantics which yields a family of queries equivalent to the computable queries.

1 Introduction

In the database community over the past decade there has been wide interest in hierarchical database structures. Of particular interest are those constructed using the tuple and set constructs; these have been studied variously as complex objects [AB88, BK89, Hul87], nested or non-first normal form relations [FT83, RKS88, JS82], and also arise in the semantic [AH87, HM81, HK87, Shi81] and other database models [KV84], etc. Recent work on query languages for complex objects has focused on developing [AB88, KV84, RKS88] and studying [KV88, GvG88] natural extensions of the relational calculus and algebra to use the set construct. All of these papers have

*This work supported in part by NSF grants IST-85-11541 and IRI-87-19875 and is based largely on an extended abstract which appeared in "ACM Symp. on Principles of Database Systems, March, 1988."

†Current address: Computer Science Department, University of California, Santa Barbara, CA 93106.

arrived at a calculus language in the spirit of Jacobs' database logic [Jac82], and a corresponding algebra which has equivalent expressive power. In this paper we initiate an investigation into how the use of sets in these languages extends their expressive power, and increases the complexity of computing answers. In particular, we provide basic results regarding the expressive power and complexity of different subsets and variations of the complex object calculus (and algebra).

A primary focus in this paper is on queries which map flat relations to flat relations. From this perspective, we are studying a family of natural extensions of the relational calculus, in the general spirit of Chandra and Harel [CH82], and DATALOG. In this paper, we establish a hierarchy of queries which lies between the second order queries (SO) of [CH82] and the computable queries of [CH80, AV87]. We thus provide a framework for analyzing query languages for the relational model whose expressive power stands above SO. We also consider queries from the complex object calculus which use *invented values* as "scratch paper". This is related to the 'with new' construct used in connection with procedural languages in [AV87] and to the creation of new object identifiers supported in many object-oriented database models. This yields a family of languages, some equivalent to the computationally 'complete' languages of [CH80, AV87, AV88], and others more powerful.

Speaking informally, the types in our model can be viewed as trees which are constructed recursively using the tuple (or Cartesian product) construct, and the (finitary) set construct. The *set-height* of a type is defined to be the maximum number of set constructs in any path of the type. Thus, all variables in a query in the classical relational calculus range over objects of types with set-height 0. Results of [Var82, AB88] show that there is a query in the complex object calculus which computes the transitive closure of a binary relation. This query uses a variable whose type has set-height 1. Because the transitive closure cannot be computed in the relational calculus [AU79], this demonstrates the fact that using *intermediate types* with set-height > 0 yields increased expressive power in a calculus-based language.

In this paper we define $\mathrm{CALC}_{k,i}$ ($\mathrm{ALG}_{k,i}$) to be the set of calculus (algebraic) queries mapping from and to instances with set-height $\leq k$ and using intermediate types with set-height $\leq i$. In particular, for each i, $\mathrm{CALC}_{0,i}$ is the family of calculus queries mapping from and to the relational model, and using intermediate types with maximum set-height i. Thus, transitive closure is in $\mathrm{CALC}_{0,1} - \mathrm{CALC}_{0,0}$. Also, it is easily seen that $\mathrm{CALC}_{0,1}$ is equivalent in expressive power to the second order queries SO of [CH82]. We show here that the (data) complexity [Var82] of $\mathrm{CALC}_{0,i}$ ($i \geq 1$) is bounded below by $(i-1)$-level hyper-exponential time, and above by $(i-1)$-level hyper-exponential space. Furthermore, $\mathrm{CALC}_{0,i}$ is able to express all (generic) queries from flat databases to flat relations whcih are computable in $(i-1)$-level hyper-exponential time. In fact, $\cup_{i \geq 0} \mathrm{CALC}_{0,i}$ is equivalent in expressive power to the class of *elementary queries*, i.e., the class of generic database mappings in the relational model computable in hyper-exponential time (or space). Finally, we show that the $\mathrm{CALC}_{0,i}$ hierarchy does not collapse because for each $i \geq 0$, $\mathrm{CALC}_{0,i}$ is strictly weaker than $\mathrm{CALC}_{0,i+1}$.

The proof of this is based on a spectra theorem of [Ben62].

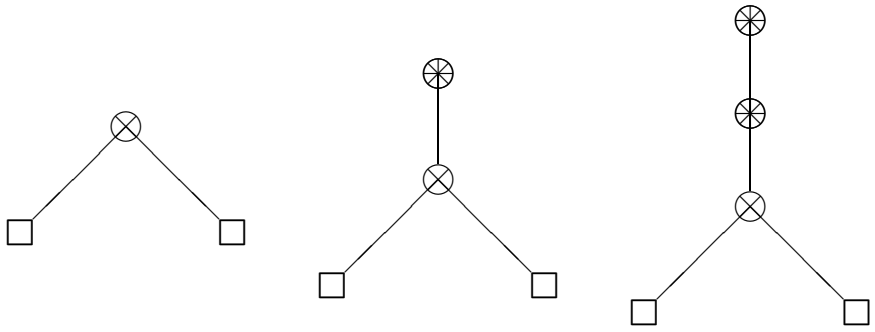
The study of the use of intermediate types just described is closely related to an independent investigation by Kuper and Vardi [KV88]. They characterize the complexity of families of queries (with multiple levels of nested sets, similar to the $\text{CALC}_{0,i}$ families) for the Logical Data Model [KV84] using alternating Turing machines with hyper-exponential running time.

Speaking intuitively, one of the ways which intermediate types can be "used" by queries is to provide large sets of indices for arrays or other data structures, e.g., to store the encoding of a Turing machine computation. We introduce several semantics for the calculus based on invented values to isolate this "use" from other uses. (One of these turns out to be equivalent to the *unlimited interpretation* [Mai83]). Using these alternative semantics the $\text{CALC}_{0,i}$ hierarchies collapse at $\text{CALC}_{0,1}$. Furthermore, under these semantics there is a 'universal type' T_{univ} which can be used to encode objects of all types. (It is shown elsewhere [AGSS86, HS88, HS89a] that these semantics do not increase the expressive power of the relational calculus.) Also, under these semantics the family $\text{CALC}_{0,1}$ is more powerful than the computationally 'complete' languages of [CH80, AV87, AV88]; in particular, there is a query in these languages which specifies a total mapping which corresponds to the halting problem. We also present yet another semantics based on invention which yields a computationally complete language.

The organization of this paper is as follows. In Section 2 we provide the definitions needed for our study, including those of (complex) type, object, and instance; and the calculus and algebra query languages for them. Section 3 formally introduces the notion of intermediate types in queries and presents some basic results concerning them. In particular, we note that whenever $0 \leq k \leq i$, $\text{CALC}_{k,i}$ is equivalent in expressive power to $\text{ALG}_{k,i}$; and that $\text{CALC}_{0,1}$ is equivalent to SO. Section 4 focuses on the complexity of evaluating queries in $\text{CALC}_{0,i}$, and presents upper and lower space- and time-bounds in complexity and expressive power. It also defines the basic notion of *elementary queries* which characterizes the $\text{CALC}_{0,i}$ hierarchy, and briefly mentions the correspondence of the results in sections 4 and 5 and the results of [KV88]. Section 5 demonstrates that each level of the $\text{CALC}_{0,i}$ hierarchy has more expressive power than the previous one. Section 6 introduces and studies the semantics based on invented values.

2 Preliminaries

In this section we introduce the basic framework for our investigation, including definitions of complex types, objects and instances; and the complex object calculus and algebra. In this framework, types are built from set and tuple constructs. The calculus and algebra are similar to that in [AB88], but do not include "built-in" (or "user defined") predicates or functions. Due to the particular focus of the investigation, we use a somewhat restricted notion of complex object, but it will be clear that our results apply to more general classes of types built from tuple and set. Finally, the semantics presented here is generalized to permit the study of several classes of database



$T_1 = [U, U]$ (a) $T_2 = \{[U, U]\}$ (b) $T_3 = \{\{[U, U]\}\}$ (c)

Figure 1: Three types

queries.

We assume the existence of a countably infinite *universal domain* **U** of *atomic objects*. Types are built recursively using the symbol 'U' and the tuple and set constructs:

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

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

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

(c) if $T_1, ..., T_n, n \geq 1$ are basic and/or set types then $[T_1, ..., T_n]$ is a *tuple* type.

Types can be viewed as trees (whose nodes have ordered children) with leaf nodes corresponding to the basic type and internal nodes corresponding to the set and tuple constructs. The types we define here are essentially equivalent to the non-first normal form relations of [Mak77, JS82, FT83], the complex objects of [AB88], and the (object) types of the IFO model [AH87] (and from other semantic data models). They are also essentially equivalent to the formats of [HY84] except that the marked union (or generalization) construct is not included here. Finally, there is a close relationship between the types here and the Logical Data Model of [KV84].

Example 2.1: Figure 1 shows three types and their natural tree representations. □

Note that in this definition we do not permit consecutive application of the tuple construct. Sometimes it is convenient to build "types" involving the consecutive application of tuple. There is a natural 'collapse' transformation of these "types" into types as formally defined above, which preserves information capacity [HY84], etc. For the purposes of this paper we will sometimes discuss a "type" T with consecutive application of tuple; in the formal development we mean that the collapse of T is used.

With each type we associate a set of 'objects' having the "shape" proscribed by the type.

Definition: For each type T the *domain* of T, denoted $dom(T)$, is defined recursively as follows:

(a) if $T = U$ then $dom(T) = \mathbf{U}$;

(b) if $T = \{T_1\}$ then $dom(T) = \mathcal{P}^{\text{fin}}(dom(T_1)) = \{Y \mid Y \subseteq dom(T_1) \text{ and } Y \text{ finite }\}$;

(c) if $T = [T_1, ..., T_n]$ then $dom(T) = dom(T_1) \times \cdots \times dom(T_n)$.

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

Note that each instance of a type T is an object of type $\{T\}$.

Example 2.2: Referring to Figure 1, if {Tom, Mary, Sue, ...} $\subseteq \mathbf{U}$ then [Tom, Mary] $\in dom(T_1)$, and {[Tom, Mary], [Mary, Sue]} is both an instance of T_1 and an object of T_2. Thus, $dom(T_2)$ consists of all finite binary relations over $\mathbf{U}$. An object of T_3 is a finite set of objects of T_2, i.e., a finite set of binary relations. □

A crucial characteristic of types is the level of nested set constructs it permits:

Definition: The *set-height* of a type T, denoted $sh(T)$, is defined to be the maximum number of set nodes in a path of T from root to leaf. For $i \geq 0$, τ_i denotes the set $\{T \mid T \text{ is a type and } sh(T) = i\}$.

Example 2.3: In the Example 2.1, $sh(T_1) = 0$; $sh(T_2) = 1$; and $sh(T_3) = 2$. □

Note that the family τ_i, $i \geq 0$, imposes a partition on the family of types. Also, each relation in the relational model can be viewed as an instance of a type in τ_0, and each type in τ_0 corresponds to a relation schema. Finally, each variable in a relational calculus query ranges over tuples, i.e., over objects with type in τ_0.

Following the relational model, [AB88], and others, we view a database to be a finite sequence of instances:

Definition: Let $\mathbf{P}$ be a countably infinite set of abstract *predicate names*. A *database schema* is a sequence $D =< P_1 : T_1, \ldots, P_n : T_n >$ where

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

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

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

An *instance* of D is a sequence $d =< P_1 : I_1, \ldots, P_n : I_n >$ where I_j is an instance of T_j for each $j \in [1..n]$. The family of instances of D is denoted $inst(D)$.

In this paper, we are concerned primarily with queries from conventional databases to conventional relation instances. To this end, we introduce:

Definition: A type T (database schema D) is *flat* if the set-height of T (each type in D) is 0.

This concludes the definitions of the kinds of objects found in our model. Before moving to the query languages we introduce the roughly dual notions of 'active domain' and 'constructive domain'. The active domain of an object (type instance, database instance) is the set of atomic objects occurring "in" the object (type instance, database instance):

Definition: For a type T and object $X \in dom(T)$, the *active domain* of X, denoted $adom(X)$ is defined recursively as:

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

(b) if $T = \{T_1\}$ then $adom(X) = \bigcup\{adom(Y) \mid Y \in X\}$; and

(c) if $T = [T_1, ..., T_n]$ and $X = [X_1, ..., X_n]$ then $adom(X) = \bigcup\{adom(X_i) \mid i \in [1..n]\}$.

The *active domain* of an instance I of T is $adom(I) = \bigcup\{adom(X) \mid X \in I\}$. The *active domain* of a database instance $d = \langle P_1 : T_1, ..., P_n : T_n\rangle$ is $adom(d) = \bigcup\{adom(I_j) \mid j \in [1..n]\}$.

Active domain can be viewed as a mapping from complex objects to subsets of **U**. Intuitively, something like an inverse of this is provided by:

Definition: Let T be a type and Y a subset of **U**. The *constructive domain* of T from Y is $cons_Y(T) = \{X \mid X \in dom(T)$ and $adom(X) \subseteq Y\}$.

Several articles have introduced calculus and algebra query languages for complex objects based on the tuple and set constructs [AB88, Jac82, KV84, RKS88]. In what follows we use a syntactically simple calculus and algebra which have the same expressive power (except without "built-in" functions as in [AB88]). In particular, the calculus has the ability to use variables ranging over types of arbitrary set-height; and the algebra includes the powerset operator. Importantly, both the calculus and algebra are strongly typed.

We now introduce the calculus for complex objects. Definitions are given after the following informal description. Briefly, the 'terms' of our calculus are constants, variables, and expressions of the form $x.i$ where x is a variable and $i \in \mathbf{N}$. The 'atomic formulas' are (typed) expressions of the form $t_1 \approx t_2, t_1 \in t_2$, and $P(t_1)$, where t_1, t_2 are terms and P is a predicate symbol. Formulas are built using the sentential connectives in the usual manner, and with 'typed' quantifiers ($(\forall x/T\, \phi)$ and $(\exists x/T\, \phi)$), similar in spirit to the approach taken by Reiter [Rei80]. Speaking intuitively, a typed quantifier of a variable can be viewed as *assigning* a type to the occurrences of that variable within the scope of the quantifier. Finally, a 'typed calculus query' from a database schema $D = \langle P_1 : T_1, ..., P_n : T_n\rangle$ to a type T is an expression $Q = \{t/T \mid \phi\}$ where ϕ is built

from the predicate symbols in D, has only t free, and satisfies the natural constraints concerning typing. The application of Q on a database instance d is denoted $Q[d]$.

The language for formulas in the calculus language uses the following symbols:

1. constant symbols, which are members of **U**;
2. predicate symbols, which are members of **P**;
3. coordinate specifiers, which are elements of **N** (the natural numbers);
4. the symbols for type expressions, namely U, {, }, [,], and ,;
5. variable symbols, generally taken from t, u, v, w, x, y, z (possibly with subscripts or primes);
6. equality and element-of symbols $\approx$ and $\in$;
7. sentential connectives $\neg, \wedge, \vee, \rightarrow$;
8. quantifiers $\forall, \exists$; and
9. punctuation symbols (,), ., /.

In the following, we first introduce the notions of terms, well-formed-formulas, and queries, i.e., the syntax of a calculus query, and then describe the associated semantics. Since the variables and predicate symbols are typed in well-formed-formulas and queries, we need to have the concept of assigning types to variables and predicates. To describe formally this assignment of types, we define a 'typed formula' to be a pair (ϕ, α) where ϕ is a formula and α is an assignment of types to variables and predicates in ϕ which satisfies certain constraints. The following straightforward but tedious formal development is included for completeness.

A *type assignment* is a partial mapping from variables and predicates to types. If α is a type assignment, a *term* under α in the calculus is

(a) a constant symbol;

(b) a variable symbol x if $\alpha(x)$ is defined; or

(c) the expression $x.i$ if x is a variable, $\alpha(x) = [T_1, ..., T_n]$ is a tuple type, and $i \in [1..n]$.

We do not need terms of the form $x.i.j$ because our formal definition of types does not permit consecutive application of the tuple construct.

Let α be a type assignment. The *extended type assignment* of α, denoted $\overline{\alpha}$, is a partial mapping from terms and predicates to types defined as follows:

$$\overline{\alpha}(t) = \begin{cases} U & \text{if } t \in \mathbf{U} \text{ is a constant,} \\ \alpha(t) & \text{if } t \text{ is a variable or predicate,} \\ T_i & \text{if } t = x.i \text{ and } \alpha(x) = [T_1, ..., T_n]. \end{cases}$$

If α_1, α_2 are two type assignments, we denote $\alpha_1 \cup \alpha_2$ (union) as a new type assignment defined from α_1 and α_2:

$$\alpha_1 \cup \alpha_2(t) = \begin{cases} \alpha_1(t) & \text{if } \alpha_2(t) \text{ is not defined,} \\ \alpha_2(t) & \text{if } \alpha_1(t) \text{ is not defined,} \\ \alpha_1(t) & \text{if } \alpha_1(t) \text{ and } \alpha_2(t) \text{ are both defined and } \alpha_1(t) = \alpha_2(t), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We also denote $\alpha_1(x/T)$ (substitution) as: $\alpha_1(x/T)(x) = T$ and $\alpha_1(x/T)(t) = \alpha_1(t)$ for $t \neq x$, and $\alpha_1 \uparrow x$ (undefine) as: $\alpha_1 \uparrow x(t) = \alpha_1(t)$ for $t \neq x$ and $\alpha_1 \uparrow x(x)$ undefined.

Definition: The family of *typed-well-formed-formulas* (or *t-wffs*) of the calculus is a set of pairs of formula and type assignment defined recursively by:

1. *Atomic* formulas. If t_1, t_2 are two terms under type assignments α_1, α_2 respectively and P is a predicate, then
   (a) $((t_1 \approx t_2), \alpha_1 \cup \alpha_2)$ is a t-wff if $\overline{\alpha}_1(t_1), \overline{\alpha}_2(t_2)$ are defined, $\overline{\alpha}_1(t_1) = \overline{\alpha}_2(t_2)$, and $\alpha_1 \cup \alpha_2(x)$ is defined for each variable x occurring in t_1 or t_2;
   (b) $((t_1 \in t_2), \alpha_1 \cup \alpha_2)$ is a t-wff if $\overline{\alpha}_1(t_1)$, $\overline{\alpha}_2(t_2)$ are defined, $\overline{\alpha}_2(t_2) = \{\overline{\alpha}_1(t_1)\}$, and $\alpha_1 \cup \alpha_2(x)$ is defined for each variable x occurring in t_1 or t_2; and
   (c) $(P(t_1), \alpha_1)$ is a t-wff if $\overline{\alpha}_1(t_1)$, $\overline{\alpha}_2(t_2)$ are defined and $\overline{\alpha}_1(P) = \overline{\alpha}_1(t_1)$.
2. If (ϕ_1, α_1) and (ϕ_2, α_2) are t-wffs then $(\neg\phi_1, \alpha_1)$ is a t-wff and $((\phi_1 \wedge \phi_2), \alpha_1 \cup \alpha_2)$, $((\phi_1 \vee \phi_2), \alpha_1 \cup \alpha_2)$, and $((\phi_1 \rightarrow \phi_2), \alpha_1 \cup \alpha_2)$ are t-wffs provided that $\alpha_1 \cup \alpha_2(x)$ is defined for each variable x occurring free in ϕ_1 or ϕ_2.
3. If (ϕ, α) is a t-wff, x a variable, T a type, and either x is not a free variable in ϕ or $\alpha(x) = T$, then $((\forall x/T\, \phi), \alpha \uparrow x)$ and $((\exists x/T\, \phi), \alpha \uparrow x)$ are t-wffs.

From the definition, it is easy to see that if (ϕ, α) is a t-wff and x is a free variable in ϕ then $\alpha(x)$ is defined. We now define a query to be a syntactic object which consists of a target variable, an associated type, and a t-wff.

Definition: If $D = \langle P_1 : T_1, \ldots, P_n : T_n \rangle$ is a database schema and (ϕ, α) is a t-wff, then (ϕ, α) is a *query-formula* from D if

(a) the set of predicate symbols occurring in ϕ is contained in $\{P_1, \ldots, P_n\}$; and

(b) $\alpha(P_i) = T_i$ for $i \in [1..n]$.

Let T be a type. A *typed calculus query* from D to T is an expression $Q = \{t/T \mid \phi\}$ where

(a) the set of variables occurring free in ϕ is contained in $\{t\}$;

(b) (ϕ, α) is a query-formula from D where $\alpha(t) = T$ and the domain of α is $\{t, P_1, ..., P_n\}$.

In this case we write $Q : D \to T$.

Finally, the *active domain* of a formula ϕ (query Q), denoted $adom(\phi)$ ($adom(Q)$), is the set of atomic values occurring in ϕ (Q). This concludes the presentation of the syntax of the complex object calculus.

For notational convenience, in this paper we also use various shorthands for the calculus. For example, the tuple construct $[y, y]$ stands for $\exists z/[T, T]$ $(z.1 \approx y \wedge z.2 \approx y)$ within the scope of a quantifier for y of type T; and $x \approx \emptyset$ for $\forall y$ $(y \notin x)$.

We now turn to the semantics of queries. Roughly speaking, this paper is divided into two parts. In the first part (sections 3 to 5), the usual semantics – the limited interpretation (or the active domain interpretation), is used. In the second part (section 6), we shall study several alternative semantics for the same calculus. These kinds of semantics, in essence, are based on "invented values". For technical convenience, we now provide the following general definition of semantics for calculus queries.

Let α be a type assignment and X a subset of $\mathbf{U}$. A *value assignment* under α and X is a partial mapping ρ from variables to objects such that for each variable x, $\rho(x) \in cons_X(\alpha(x))$ if $\alpha(x)$ is defined. Similar to the type assignment, we define *extended value assignment* and *substitution*.

In the following definition of satisfaction, the set Y is used to control the range of quantified variables.

Definition: Suppose now that $D = \langle P_1 : T_1, ..., P_n : T_n \rangle$ is a database schema and $d = \langle P_1 : I_1, ..., P_n : I_n \rangle$ is an instance of D. Suppose further that (ϕ, α) is a query-formula, and Y a subset of $\mathbf{U}$. If ρ is a value assignment under α and $X = Y \cup adom(d) \cup adom(\phi)$, then d *satisfies* (ϕ, α) (ϕ if α is understood) under ρ and Y, denoted $d \models_Y (\phi, \alpha)[\rho]$ ($d \models_Y \phi[\rho]$) or $d \models (\phi, \alpha)[\rho]$ ($d \models \phi[\rho]$) when $Y = \emptyset$, if one of the following is true:

1. $\phi = (t_1 \approx t_2)$ and $\overline{\rho}(t_1) = \overline{\rho}(t_2)$;
2. $\phi = (t_1 \in t_2)$ and $\overline{\rho}(t_1) \in \overline{\rho}(t_2)$;
3. $\phi = P_i(t)$ and $\overline{\rho}(t) \in I_i$;
4. $\phi = \neg\phi_1$ and $d \not\models_Y (\phi_1, \alpha)[\rho]$;
5. $\phi = (\phi_1 \wedge \phi_2)$ $((\phi_1 \vee \phi_2)$, $(\phi_1 \to \phi_2))$, and that $d \models_Y (\phi_1, \alpha)[\rho]$ and (or, implies) $d \models_Y (\phi_2, \alpha)[\rho]$;
6. $\phi = (\exists x/T\, \phi_1)$ and $d \models_Y (\phi_1, \alpha(x/T))[\rho(x/o)]$ for some object $o \in cons_X(T)$; or
7. $\phi = (\forall x/T\, \phi_1)$ and $d \models_Y (\phi_1, \alpha(x/T))[\rho(x/o)]$ for every object $o \in cons_X(T)$.

If $Q = \{t/T \mid \phi\}$ a calculus query from D to T, then

$$Q|^Y[d] = \{o \in cons_X(T) \mid d \models_Y (\phi, \alpha_{D,Q})[\rho(t/o)] \text{ for some } \rho\}.$$

Now, the definition of the limited interpretation is given:

Definition: If D is a database schema, d an instance of D, and Q is a query from D, then the answer of Q on d under the *limited interpretation* is $Q[d] = Q|^{\emptyset}[d]$.

Intuitively, under the limited interpretation, all variables range over objects constructed out of the active domain of the (current) database instance and query. In the context of the relational model, this is the *limited interpretation* of [Mai83]. Following [AB88], a query Q is *domain independent* if for each set $Y \subseteq \mathbf{U}$, $Q|^{Y}$ and $Q|^{\emptyset}$ define the same mapping. This is closely related to the notion of *safe* [Mai83, Ull82]. The *conventional* (or *underlying domain*) interpretation of a query Q is given by $Q|^{\mathbf{U}}$. It is known that the family of safe relational calculus queries under the conventional interpretation has expressive power equal to that of the family of arbitrary relational calculus queries under the limited interpretation. This is naturally generalized to the complex object calculus (Proposition 3.3). For technical convenience, in sections 3 to 5, we use the limited interpretation.

Example 2.4: Consider the database $D = \langle \text{PAR: } T_1 \rangle$, where T_1 is the type $[U, U]$ of Example 2.1. Let $d = \langle \text{PAR: } I_1 \rangle$ be an instance of D. The query $Q_1 = \{t/T \mid \psi(t)\}$ where

$$\psi(t) = \exists x/T_1 \, \exists y/T_1 \, (\text{PAR}(x) \wedge \text{PAR}(y) \wedge x.2 \approx y.1 \wedge t.1 \approx x.1 \wedge t.2 \approx y.2)$$

is a typed query which computes $\pi_{1,4}$ (PAR $\bowtie_{2=3}$ PAR). (If PAR holds the parent relation, then $Q_1[d]$ yields the grandparent relation).

Let T_2 be the type $\{[U, U]\}$ in Example 2.1 and $Q_2 = \{t/T_2 \mid \phi(t)\}$ where

$$\begin{aligned}
\phi(x) \;=\; & \forall y/T_1(y \in x \rightarrow \exists z/T_1(\text{PAR}(z) \wedge (y.1 \approx z.1 \vee y.1 \approx z.2)) \wedge \\
& \quad \exists z/T_1(\text{PAR}(z) \wedge (y.2 \approx z.1 \vee y.2 \approx z.2))) \\
& \wedge \forall y/T_1(\text{PAR}(y) \rightarrow y \in x) \\
& \wedge \forall y/T_1 \forall y'/T_1((y \in x \wedge y' \in x \wedge y.2 \approx y'.1) \rightarrow \\
& \quad \exists y''/T_1(y'' \in x \wedge y''.1 \approx y.1 \wedge y''.2 \approx y'.2)).
\end{aligned}$$

Then Q_2 is a typed query which maps $\langle \text{PAR: } T_1 \rangle$ to T_2 and $Q_2[d]$ is the set of all binary relations R such that $adom(R) \subseteq adom(I_1)$; $I_1 \subseteq R$; and R is transitive. Note that the transitive closure of I_1 is *one* of the elements of $Q_2[d]$. $\square$

Although our main focus is the complex object calculus, our results also apply directly to complex object algebras (see Theorem 3.8). The algebra used here is essentially equivalent to those of [AB88, KV84, RKS88]. Algebra expressions are built from predicate symbols, constants, and algebraic operators. Each algebra expression E has an associated type $\overline{\alpha}(E)$, where α assigns types to predicate symbols, and E has range $inst(\overline{\alpha}(E))$, i.e., all instances of that type.

Definition: Let α be a type assignment. The family of (*typed*) *algebraic expressions* and their corresponding types is defined recursively by: E is an algebraic expression of type $\overline{\alpha}(E)$ if one of the following is true:

(1) $E = P$, $P \in \mathbf{P}$, and $\overline{\alpha}(E) = \alpha(P)$;

(2) $E = \{a\}$, $a \in \mathbf{U}$, and $\overline{\alpha}(E) = U$;

or if E_1, E_2 are algebraic expressions of types $\overline{\alpha}(E_1)$, $\overline{\alpha}(E_2)$, then:

(3) $E = (E_1 \cup E_2)$ (or $(E_1 \cap E_2)$, $(E_1 - E_2)$), $\overline{\alpha}(E_1) = \overline{\alpha}(E_2)$, and $\overline{\alpha}(E) = \overline{\alpha}(E_1)$;

(4) $E = \pi_{i_1,...,i_k}(E_1)$, $\overline{\alpha}(E_1) = [T_1, ..., T_n]$, $i_1, ..., i_k \in [1..n]$, and $\overline{\alpha}(E) = [T_{i_1}, ..., T_{i_k}]$;

(5) $E = \sigma_F(E_1)$, $\overline{\alpha}(E_1) = [T_1, ..., T_n]$, $\overline{\alpha}(E) = \overline{\alpha}(E_1)$, 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 $t_i \in [1..n]$ and obeying the natural typing requirements; e.g., if $\alpha(T_1) = \{\alpha(T_2)\}$ then $1 \in 2$ is permitted but $1 = 2$ is not;

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

(6) $E = E_1 \times E_2$ and $\overline{\alpha}(E) = [f(\overline{\alpha}(E_1)), f(\overline{\alpha}(E_2))]$ where f maps any type to a sequence of types: $f(U) = U$, $f(\{T\}) = \{T\}$, and $f([T_1, ..., T_n]) = T_1, ..., T_n$;

(7) $E = \mathcal{U}(E_1)$ (untuple), $\overline{\alpha}(E_1) = [T]$, and $\overline{\alpha}(E) = T$;

(8) $E = \mathcal{C}(E_1)$ (collapse), $\overline{\alpha}(E_1) = \{T\}$ for some type T, and $\overline{\alpha}(E) = T$; or

(9) $E = \mathcal{P}(E_1)$ (powerset) and $\overline{\alpha}(E) = \{\overline{\alpha}(E_1)\}$.

Now let $D = \langle P_1 : T_1, ..., P_n : T_n \rangle$ be a database schema. An *algebraic query expression* is a typed algebraic expression E for type assignment α, where $\alpha(P_i) = T_i$ and undefined elsewhere, such that every predicate symbol occurring in E is in $\{P_1, ..., P_n\}$.

Since the main focus is the calculus, we describe the semantics associated with algebraic queries only briefly. If d is an instance of D, then the semantics of (1) to (6) is defined in the usual manner; the semantics of (7) is to remove the topmost tuple construct (its inverse can be obtained by $\pi_1(E \times E)$); (8) $\mathcal{C}(E)[d] = \bigcup\{x \mid x \in E[d]\}$; and (9) $\mathcal{P}(E)[d] = \{x \mid x \subseteq E[d]\}$.

It should be noted that the non-first normal form relational operators nest and unnest can be simulated using various combinations of the above operators. To compare the expressive power of query languages we need:

Definition: Let Q_1, Q_2 be two queries (possibly from different query languages). Then Q_1 and Q_2 are *equivalent*, denoted $Q_1 \equiv Q_2$, if

(a) Q_1 and Q_2 map from the same database schema;

(b) Q_1 and Q_2 map to the same type; and

(c) $Q_1[d] = Q_2[d]$ for each instance $d \in inst(D)$.

Definition: If C, C' are two families of queries, C is *no more expressive* than C', denoted $C \sqsubseteq C'$, if for each $Q \in C$, there is a $Q' \in C'$ such that $Q \equiv Q'$. Further, C and C' have *equivalent expressive power*, denoted $C \equiv C'$, if $C \sqsubseteq C'$ and $C' \sqsubseteq C$. Finally, C is (*strictly*) *less expressive* than C', denoted $C \sqsubset C'$, if $C \sqsubseteq C'$ and $C \not\equiv C'$.

Definition: If $C \subseteq \mathbf{U}$, $Q : D \to T$ is a query, then: Q is C-*generic* [Hul86], if[1]

$$f \circ \sigma = \sigma \circ f \text{ for each permutation } \sigma \text{ over } \mathbf{U} \text{ with } \forall x \in C,\ \sigma(x) = x.$$

Q is *generic* if Q is C-generic for some finite C.

It is clear that each query in the calculus and algebra described here is generic.

3 Intermediate Types

In this section, we introduce the fundamental concept of *intermediate types*, illustrate their use, and state some basic results. We first present two examples to highlight one use of intermediate types. They focus on transitive closure and even cardinality recognition. A formal definition of intermediate type is then given. Two classes of families of queries, $\mathrm{CALC}_{i,j}$ and $\mathrm{ALG}_{i,j}$, are defined in terms of input-output types and intermediate types. It is noted that the two families are "almost" equivalent. Proposition 3.9 states that $\mathrm{CALC}_{0,1}$ has expressive power equivalent to the second order queries of Chandra and Harel [CH82]. We also present a result concerning the use of intermediate types in relational queries.

To motivate the discussion, we present an example for computing transitive closure which appears in [AB88] (see also [Hul87]), and an example for recognizing even cardinality (see also [CH82]).

Example 3.1: Consider the database schema $D = \langle \text{PAR: } T_1 \rangle$ (where $T_1 = [U, U]$) and formula ϕ as in Example 2.4. Let database $d = \langle \text{PAR: } I_1 \rangle$ and x be a variable of type $\{[U, U]\}$. Recall that the formula $\phi(x)$ states that each element in x is a binary relation containing I_1, transitive, and constructed from the active domain. Then the query $\{x/\{[U, U]\} \mid \phi(x)\}$ will select all binary relations over the active domain which are transitive and contain I_1. The transitive closure is now obtained by intersection:

$$Q = \{z/[U, U] \mid \Phi_{\mathrm{TC}}(z)\}$$

where $\Phi_{\mathrm{TC}}(z) = \forall x/\{[U, U]\}\ (\phi(x) \to z \in x)$. □

In the above example, the intermediate type, i.e., the type of variable x, provides much more "space" to hold temporary results which are used to obtain the final result. Transitive closure, in fact any fixpoint query, can be computed by a calculus query which uses a single existentially quantified variable with set-height 1 [Var82]. The relationship of the fixpoint operator, while loops, and the powerset operator in the context

[1] σ is extended naturally to databases.

of a complex object algebra is explored in [GvG88]. Furthermore, given instances of types in τ_0, the problem of recognizing even cardinality is easily expressed using an intermediate type from τ_1 as shown by the following example.

Example 3.2: Let the database schema $D = \langle \text{PERSON: } U \rangle$ and $d = \langle \text{PERSON: } I \rangle$. Further, let f be the database mapping defined by[2]:

$$f(d) = \begin{cases} I & \text{if } |I| \text{ is even,} \\ \emptyset & \text{if } |I| \text{ is odd.} \end{cases}$$

Using a variable x of type $\{[U, U]\}$, f can be realized by the following query Q:

$$Q = \{t/U \mid \text{PERSON}(t) \wedge \exists x/\{[U,U]\}\ (\phi_1(x) \wedge \phi_2(x)) \wedge \phi_3(x))\}$$

where

$$\begin{aligned} \phi_1(x) &= \forall y/U\ [\text{PERSON}(y) \rightarrow \exists z/[U,U]\ (z \in x \wedge [z.1 \approx y \vee z.2 \approx y])], \\ \phi_2(x) &= \forall y/[U,U]\ \forall z/[U,U]\ [y \in x \wedge z \in x \rightarrow (y.1 \approx z.1 \leftrightarrow y.2 \approx z.2)], \\ \phi_3(x) &= \forall z/U\ (\neg z \in x.1 \vee \neg z \in x.2). \end{aligned}$$

Intuitively, the formula of Q states that if there is a pairing x of all persons in the input d then output everything in PERSON. □

We now give the definition of intermediate type:

Definition: If $D = \langle P_1 : T_1, ..., P_n : T_n \rangle$ is a database schema, and $Q : D \rightarrow T$ is a query on D where $Q = \{t/T \mid \phi(t)\}$, then S is an *intermediate type* of Q, if there is a variable of type S occurring in $\phi(x)$ and $S \notin \{T_1, ..., T_n, T\}$.

From the mapping point of view, the two queries in Example 3.1 and 3.2 can be thought of as mappings on instances of types in τ_0 "using" types in τ_1 as "tools". More generally, we define the following families of queries:

Definition: $\text{CALC}_{k,i}$ is the *family* of calculus queries Q such that:

1. for some $D = \langle P_1 : T_1, ..., P_n : T_n \rangle$ and type T, $Q : D \rightarrow T$;
2. $\{T_1, ..., T_n, T\} \subseteq \bigcup_{j \leq k} \tau_j$; and
3. each intermediate type of Q is in $\bigcup_{j \leq i} \tau_j$.

From the definition, it is clear that for each k and i, (a) $\text{CALC}_{k,i} \sqsubseteq \text{CALC}_{k,i+1}$; (b) $\text{CALC}_{k,i} \sqsubseteq \text{CALC}_{k+1,i}$. Also, it is easily seen that:

Proposition 3.3: $\forall k, i \in \mathbf{N}$, if $k \leq i$ then the family of domain independent queries in $\text{CALC}_{k,i}$ under the conventional interpretation is equivalent in expressive power to the family of arbitrary queries in $\text{CALC}_{k,i}$ under the limited interpretation.

[2] $|I|$ denotes the cardinality of I

The following two examples show how Turing machine computations can be encoded using intermediate types. Recall that a computation consists of a sequence of tape configurations and positions of the tape head. Thus, in order to encode the sequence, we need to have an "order" or "index". Unfortunately, in this model, sets and the universal domain **U** are not ordered, and tuples have fixed length. So the first example below demonstrates that a "total order" can be created by using an intermediate type. The construction in this example is used several times in this paper.

Example 3.4: Suppose D is a database schema and T a type with $sh(T) = j \in \mathbf{N}$. Let

$$\begin{aligned}\mathrm{ORD}_T(x) = {} & \forall y/T\,([y,y] \in x) \wedge \\ & \forall y/T\,\forall z/T\,(y \not\approx z \rightarrow [([y,z] \in x \wedge [z,y] \notin x) \vee ([z,y] \in x \wedge [y,z] \notin x)]) \wedge \\ & \forall w/T\,\forall y/T\,\forall z/T\,([w,y] \in x \wedge [y,z] \in x \rightarrow [w,z] \in x)\end{aligned}$$

where x is a free variable of type $S = \{[T,T]\}$. Then, for each instance d of D, $d \models_\emptyset \mathrm{ORD}_T(x)[\rho]$ iff ρ maps x to some total order on $cons_{adom(d)}(T)$. Note that $sh(S) = sh(T) + 1 = j + 1$. □

With this total order, we now present the encoding of computations in Example 3.5. This encoding scheme is also used in the proofs of several results in Sections 4 and 6.

Notation: (Hyper-exponential function) $hyp(c,n,0) = n^c$ and $hyp(c,n,i+1) = 2^{c \cdot hyp(c,n,i)}$, for $i \geq 0$.

Example 3.5: Let M be a Turing machine, and T an arbitrary type. Then a variable v of type $\{[T,T,U,U]\}$ can be used to encode computations of M. In particular, a value $\overline{v}$ of v can hold a set of elements (t,p,r,s), which indicate that in the t^{th} step of the computation: the p^{th} square of M's tape has contents r; if $s = q_i$ then the tape head is on p^{th} square and current state is q_i, $s =$ "$-$" otherwise (Figure 2). The size of this encoded, two dimensional array is bounded by the number of elements in the constructive domain of T from the current active domain provided by the context. It is easy in the calculus to enforce the appropriate restrictions on v, e.g., that the first two coordinates are a key, that each consecutive pair of steps of the computation corresponds to a valid move of M, that the final state is a halting state, etc. Let $\mathrm{COMP}_{M,T}$ be the formula specifying the above constraints. Then, the following sentence is true if and only if there is a halting computation of M with running time bounded by $|cons_A(T)|$ (where A is the current active domain):

$$\exists x/\{[T]\}\ \exists y/\{[T,T,U,U]\}\ (\mathrm{ORD}_T(x) \wedge \mathrm{COMP}_{M,T}(x,y)).$$

Suppose that T is a tuple type with set-height i, and that the width of each tuple node of T is at most w. Let $a = |A|$. Then $|cons_A(T)| \leq hyp(w,a,i)$ (this bound is achieved when each branch of T has exactly set-height i, width w and the parent of each leaf is a tuple node). Thus, an increase of 1 in the set-height of T yields an exponential increase in the amount of time of a Turing machine computation which can be encoded. □

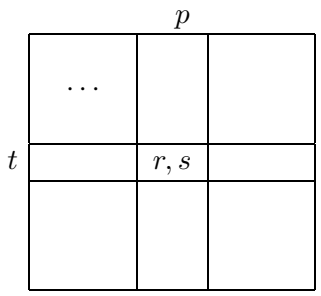


Figure 2: Encoding of Computation

Remark 3.6: It is known that if an ordering for the domain is assumed, then the language relational calculus plus fixpoint [CH82] has expressive power equivalent to PTIME queries [Var82, Imm86], and that the language relational algebra plus while [Cha81] has expressive power equivalent to PSPACE queries [Var82]. On the other hand, if an order for the domain is not given, the exact relationships between these languages and (generic) PTIME and PSPACE are not clear. The above examples show that in the context of queries which use sets, an ordering for the active domain of an input can be "created" and used in a computation. This suggests that in this context, the expressive power of various query languages will correspond exactly to (the generic subsets of) existing complexity classes. This is substantiated in part by Theorem 4.3 and Corollary 4.6. □

We also present the following example which again illustrates intuitively a correspondence between intermediate types used and the "space" available for computation. In particular, it shows how increasing the set-height of intermediate types permits the queries Q_j to use increasingly large numbers in computing an answer.

Example 3.7: Let $D =< R : U >$, $T_0 = U$ and consider the mapping f_0 from D to T_0 defined so that for $d =< R : I >$, $f_0(I) = I$ if there are numbers $p, q \leq |I|$ and ≥ 1 such that $p^4 + 1 = q^7$, and $f_0(I) = \emptyset$ otherwise. Let $S_0 = [U, ..., U]$ (width 11). Then the mapping f_0 is realized by the query

$$Q_0 = \{t/T \mid R(t) \ \wedge \exists w/\{S_0\}\, \exists x/\{U\}\, \exists y/\{U\}\, \phi(w, x, y)\}$$

where $\phi(w, x, y)$ holds iff, speaking intuitively, $x, y \subseteq I$ and w witnesses the fact that $p^4 + 1 = q^7$. More specifically, ϕ states that:

- x, y are subsets of $adom(I)$;
- w holds a 1-1 correspondence between the elements of a subset x^4 (held in $\pi_{1,...,4}(w)$) and the elements of y^7 (held in $\pi_{5,...,11}(w)$); and there is exactly one element in y^7 which does not appear in w.

More generally, consider the mapping f_j from D to T_0 defined so that for $d = < R : I >$, $f_j(d) = I$ if there are numbers $p, q \leq hyp(1, |I|, j)$ and ≥ 1 such that $p^4 + 1 = q^7$, and $f_j(I) = \emptyset$ otherwise. Let $S_j = [T_j, ..., T_j]$ (width 11), where $T_j = \{...\{U\}...\}$ (nesting j). This mapping can be realized by a query Q_j in $\text{CALC}_{0,j+1}$. This query is analogous to the query Q_0 above, except that S_j is used in place of S_0. □

The concept of intermediate type can be used for the algebra as well. Recall that each subexpression of an algebraic query is assigned a type, and that the subexpression ranges over instances of that type. The types of subexpressions are analogous to the types of variables in calculus queries. Thus *intermediate types* of an algebraic query are defined in analogy with the calculus. Finally, the families $\text{ALG}_{k,i}$ of algebraic queries are defined in the same way.

It is shown in [AB88] that the algebra is expressively equivalent to the calculus. Carefully examining their reductions in the proof, it is easy to obtain correspondent equivalences between families of calculus and algebraic queries mentioned above when $i \geq k$. A subtlety arises when $i < k$. For example, the calculus query $\{t/U \mid t \approx t\}$ always returns the acitve domain of the input. (This results from our choice to use the semantics of the limited interpretation, and is not true under the conventional semantics.) However, in the algebra one has to break down the complex structures to get the same answer. Thus the restriction $i \geq k$ is needed for the direction $\text{CALC}_{k,i} \sqsubseteq \text{ALG}_{k,i}$.

Theorem 3.8: $\forall k, i \in \mathbf{N}$, if $i \geq k$ then $\text{ALG}_{k,i} \equiv \text{CALC}_{k,i}$. □

For the remainder of this paper, we focus primarily on the families of queries $\text{CALC}_{0,i}$. In this manner, we can study the power which the complex object approach brings to queries mapping from relations to relations. The following easily verified theorem establishes the relationship between $\text{CALC}_{0,1}$ and the language of second order queries SO of [CH82].

Proposition 3.9: $\text{SO} \equiv \text{CALC}_{0,1}$. □

By the above proposition and results from [AU79, CH82], it is easily concluded that $\text{CALC}_{0,0} \sqsubset \text{CALC}_{0,1}$. Another interesting family of queries is the set $\text{DATALOG}^\neg$ of queries defined by stratified datalog programs [ABW88, vG88]. Results of [CH85, Kol87] imply that $\text{DATALOG}^\neg \sqsubset \text{CALC}_{0,1}$. Finally, is was observed that SO essentially expresses the polynomial hierarchy [Sto77].

The final result of this section concerns intermediate types in queries from the relational calculus. These arise from the use of tuple variables whose arity is not equal to the arity of the input or output relations. The following theorem shows that these intermediate types do not yield increased expressive power. In order to obtain this result, we need the following lemma which allows us to transform queries without intermediate types into a certain form, based on which the proof of Theorem 3.11 is easily

carried out (this proof is reminiscent to the proof that domain calculus can simulate tuple calculus [Ull82]).

Lemma 3.10: For any query $Q = \{t \mid \phi(t)\} \in \text{CALC}_{0,0}$ which does not use intermediate types, there exists an equivalent query Q' which also has no intermediate types, such that:

$$Q' = \{t \mid \exists v_1 \cdots \exists v_k \ (\phi'(v_1, ..., v_k) \wedge (\bigwedge_{i=1}^{k} t.i \approx v_i.1))\}$$

where k is the arity of t and for each $i \in [1..k]$, the arity of v_i is the same as that of some relation schema.

Proof: Let $\{v_1, ..., v_k\}$ be the set of variables all of the same arity of some relation schema in the database. The idea of the proof is to replace all occurrences of $t.i$ by $v_i.1$ in the formula $\phi(t)$. □

Theorem 3.11: For each query $Q \in \text{CALC}_{0,0}$, there exists another query $Q' \in \text{CALC}_{0,0}$ such that

(a) $Q \equiv Q'$; and

(b) Q' does not have any intermediate types.

Proof: Since relational algebra and calculus have the same expressive power we can easily find an algebraic query E which is equivalent to Q. Now we prove by induction that all algebraic queries have equivalent calculus queries which do not use intermediate types.

Basis: Either $E = R$ or $E = \{c\}$ where R is a relation schema and c is a constant. It is obvious that no variables are necessary to be introduced.

Induction: For the cases $E = E_1 \cup E_2$, $E = E_1 - E_2$, or $E = \sigma_F(E_1)$, the construction of $Q = \{t \mid \phi(t)\}$ is straight forward as in any usual reduction where $\phi(t) = \phi_1(t) \vee \phi_2(t)$, $\phi(t) = \phi_1(t) \wedge \neg\phi_2(t)$, or $\phi(t) = \phi_1(t) \wedge F'(t)$ respectively.

Now in the cases where $E = E_1 \times E_2$ or $E = \Pi_{i_1 \cdots i_p}(E_1)$ the query (queries) Q_1 (and Q_2) is normalized first according to Lemma 3.10. And then the equivalent calculus queries are expressed as:

$$Q = \{t \mid \exists u_1 \cdots \exists u_{k_1} \exists v_1 \cdots \exists v_{k_2} \ (\phi_1'(u_1, ..., u_{k_1}) \wedge \phi_2'(v_1, ..., v_{k_2}) \wedge \psi)\}$$

or

$$Q = \{t \mid \exists v_1 \cdots \exists v_{k_1} \ [\phi_1'(v_1, ..., v_{k_1}) \wedge (\bigwedge_{j=1}^{p} t.j = v_{i_j}.1)]\}$$

where $k_1(k_2)$ is the arity of $E_1(E_2)$ and $\psi = (\bigwedge_{i=1}^{k_1} t.i = u_i.1) \wedge (\bigwedge_{i=1}^{k_2} t.(k_1 + i) = v_i.1)$. □

4 Complexity

This section focuses on complexity issues of queries in $\mathrm{CALC}_{0,i}$, specifically (data) complexity in the sense of [Var82]. A main result of the section, Corollary 4.6, shows that the $\mathrm{CALC}_{0,i}$ hierarchy is equivalent to the family $\mathcal{E}$ of all generic database mappings computable in hyper-exponential time (space). The different levels of the $\mathrm{CALC}_{0,i}$ hierarchy are naturally embedded in the hierarchies of hyper-exponential time and space complexity classes. We indicate at the end of this section the relationship between our investigation and the related, independent investigation reported in [KV88]. The section also reviews known results concerning the lowest classes of the $\mathrm{CALC}_{0,i}$ hierarchy. For clarity and readability, we permit "types" which have consecutive tuple constructs. It is clear that collapsing them will not change the results presented.

Definition: Let Q be a query and t: $\mathbf{N} \to \mathbf{N}$ be a function. Q has time (space) complexity t, if there exists a Turing machine M which decides $o \in Q[d]$ within time (space)[3] $t(\|d\| + \|o\|)$ whenever d is an input database instance and o is an object of the target type.

We begin the discussion by reviewing known results concerning $\mathrm{CALC}_{0,0}$ and a natural subclass of $\mathrm{CALC}_{0,1}$. First, we have:

Theorem 4.1: [Var82] Each query in $\mathrm{CALC}_{0,0}$ has space complexity $O(\log n)$. □

In [Var82], Vardi introduced a language SF, which can be viewed as the relational calculus whose query formulas are of form $\exists x/\{[U, ..., U]\}\, \phi$, where ϕ does not contain any quantifiers on variables of types in τ_1. Let $\mathrm{CALC}_{0,\exists 1}$ denote the set of queries in $\mathrm{CALC}_{0,1}$ which are in prenex normal form and such that all variables of type in τ_1 are existentially quantified. The following lemma shows the equivalence between $\mathrm{CALC}_{0,\exists 1}$ and SF.

Lemma 4.2: $\mathrm{CALC}_{0,\exists 1} \equiv \mathrm{SF}$.

Proof: Obviously, $\mathrm{SF} \sqsubseteq \mathrm{CALC}_{0,\exists 1}$. We now sketch the proof for the other direction. Suppose $Q = \{t/T \mid \phi\}$ is an arbitrary query in $\mathrm{CALC}_{0,\exists 1}$. Note that ϕ is in prenex normal form. The proof is to construct a query Q' in SF equivalent to Q. The construction of Q' consists of following two key steps: (1) pushing universal quantifiers (on variables of types in τ_0) inside existential quantifiers on variables of types in τ_1; and (2) combining several consecutive existential quantifiers into one. By reasoning similar to the proof of Theorem 3.11, we assume that each intermediate type of Q which is in τ_1 is of form $\{T\}$ for some T with $sh(T) = 0$.

To show (1), let $\gamma = \forall x/T_1\ \exists y/\{T_2\}\ \psi(x, y)$ be a subformula of ϕ. It is clear that the only atomic formulas of $\psi(x, y)$ which y appears in have the form $u \approx y$ or $u \in y$.

[3] $\|d\|$ denotes the length of d. It is assumed that d and o are presented using a finite alphabet including punctuation symbols, and where, e.g., atomic elements are represented using strings over $\{0, 1\}$.

Then, γ will be replaced by the formula:

$$\exists z/\{[T_1,T_2]\}\ (\forall x/T_1\ (\exists t/[T_1,T_2]\ (t \in z \wedge x \approx t.1) \vee \psi'(x)) \wedge \forall x/T_1\ \psi''(x,z))$$

where $\psi'(x)$ is obtained from $\psi(x,y)$ through substitution of $u \approx y$ by $u \approx \emptyset$ and $u \in y$ by "false" and $\psi''(x,z)$ through substitution of $u \approx y$ by $\forall t/[T_1,T_2]\ (x \approx t.1 \rightarrow (t \in z \leftrightarrow t.2 \in u))$ and $u \in y$ by $\exists t/[T_1,T_2]\ (t \in z \wedge t.1 \approx x \wedge t.2 \approx u)$.

Repeatedly applying (1), ϕ is finally transformed to $\exists x_1/\{T_1\} \cdots \exists x_k/\{T_k\}\ \psi$ where ψ contains no quantifiers on variables of types in τ_1. In step (2), it is now easy to think of using the intermediate type $\{[T_1,...,T_k]\}$ to compress these k quantifiers. However, we must take into account the cases when $x_i = \emptyset$. But this can be checked in the calculus. We illustrate how to do this by an example when $k = 2$ and $T_1 = T_2 = U$. Then, $\phi = \exists x_1/\{U\}\, \exists x_2/\{U\}\, \psi$. ϕ will be transformed into $\exists z/\{[U,U]\}\, (\psi_{00} \vee \psi_{01} \vee \psi_{10} \vee \psi_{11})$. Intuitively, each $\psi_{i_1 i_2}$ captures one combination. For instance, ψ_{10} states the case when $x_1 \neq \emptyset$ and $x_2 = \emptyset$. ψ_{10} is actually obtained from ψ through replacing $x_1 \approx x_2, y \in x_2$ by "false" and $y \in x_1$ by $\exists t/[U,U]\, (t \in z \wedge t.1 \approx y)$.

Now, Q is equivalent to $\{t/T \mid \exists x/\{[U,...,U]\}\ \phi'\}$ where x is the only variable in ϕ' whose type is in τ_1. It is then straightforward to find the equivalent query in SF. □

Let NPTIME denote the class of problems solvable by nondeterministic Turing machines in polynomial time. Also, let QNPTIME denote the family of generic relational mappings in NPTIME.

Theorem 4.3: (a) $\text{CALC}_{0,\exists 1}$ is "complete" in NPTIME, i.e., each query in $\text{CALC}_{0,\exists 1}$ is in NPTIME, and there exists a query in $\text{CALC}_{0,\exists 1}$ which is NPTIME-complete.
(b) $\text{SF} \equiv \text{CALC}_{0,\exists 1} \equiv \text{QNPTIME}$.
(c) $\text{CALC}_{0,1}$ is at least NPTIME-hard.

Proof: Part (a) follows from Lemma 4.2 and the result of [Var82] stating that SF is complete in NPTIME. Part (b), which could be proved based on reasoning analogous to that used in the proof of Theorem 4.4 (presented shortly), is a special case of the result [Fag74] stating that the family of problems definable in existential second-order logic is equal to NPTIME. Part (c) is trivial. □

Parts (a) and (b) of the above theorem indicate the close relationship between the complexity and expressive power of $\text{CALC}_{0,\exists 1}$. As we shall see, an analogous relationship holds for the union of the $\text{CALC}_{0,i}$ hierarchy. This result is obtained by showing that the complexity and expressive power of $\text{CALC}_{0,i}$ is bounded below (above) by $(i-1)$-level hyper-exponential time (space).

Notation: Let H_i $(i \geq 0)$ be a sequence of families of functions such that $H_0 = \{\ p \mid p$ is a polynomial function from $\mathbf{N}$ to $\mathbf{N}\}$ and $H_i = \{\ 2^f \mid f \in H_{i-1}\}$ for $i \geq 1$.

Definition: Let $\mathcal{F}$ be a family of functions on $\mathbf{N}$. The family of *time restricted queries with respect to* (*wrt*) $\mathcal{F}$, denoted $\text{QTIME}(\mathcal{F})$, is defined as: $\text{QTIME}(\mathcal{F}) = \{Q \mid Q$ is a generic database mapping from flat relations to flat relations and Q has time complexity

g for some g in $\mathcal{F}\}$. The family of *space restricted queries wrt* $\mathcal{F}$, denoted QSPACE($\mathcal{F}$), is defined analogously.

The above definition is based on the (data) complexity of queries, i.e., on the complexity of checking membership of tuples in the "output". These classes can also be defined using Turing machines which compute the entire "output" [CH82], e.g., QTIME($\mathcal{F}$) = $\{Q \mid Q$ is a generic database mapping from flat relations to flat relations and Q is computable by a Turing machine operating in time g for some g in $\mathcal{F}\}$. In this paper, we are interested in QTIME(H_i), QSPACE(H_i), and in particular "elementary queries":

Definition: The family $\mathcal{E}$ of *elementary queries* is defined by: $\mathcal{E} = \bigcup_{i\geq 0}$ QTIME(H_i).

Note that $\mathcal{E} = \bigcup_{i\geq 0}$ QSPACE(H_i).

Theorem 4.4: QTIME(H_{i-1}) $\subseteq$ CALC$_{0,i}$ $\subseteq$ QSPACE(H_{i-1}) for $i \geq 1$.

Proof: We will show **(1)** CALC$_{0,i}$ $\subseteq$ QSPACE(H_{i-1}) and **(2)** QTIME(H_{i-1}) $\subseteq$ CALC$_{0,i}$.

(1) Let $i \geq 1$. It is sufficient to show that for a query $Q \in$ CALC$_{0,i}$ there is a Turing machine M which decides within space g for some $g \in H_{i-1}$ if a tuple o is in the answer of Q on a database d. We assume a Turing machine M with tape alphabet Γ which encodes atomic elements occurring in the input d and (possible) output o using strings over $\{0, 1\}$. Note that the crucial point is to have enough space to store objects $x_1, ..., x_k$ corresponding to all quantified variables of Q.

Let $\mathcal{T}$ be the set of types of variables occurring in Q, and let w be the maximum width of any tuple construct occurring within a type of $\mathcal{T}$. Let $n = ||d|| + ||o||$ be the length of the input, and $m = ||adom(d, Q)||$. Note that $m \leq n + c$ for some fixed $c \geq 0$ which depends on Q. It is easily verified for each $T \in \mathcal{T}$ with set-height i that $|cons_{adom(d,Q)}(T)| \leq hyp(w, m, i)$. Also, for $T \in \mathcal{T}$ and $o' \in cons_{adom(d,Q)}(T)$ we have:

(a) if $sh(T) = 0$, then (using the naive representation) $||o'|| \leq w \cdot m$;

(b) if $sh(T) = 1$, then $||o'|| \leq w^2 m^{w+1} \in O(hyp(w + 1, m, 0))$; and (by a straightforward induction)

(c) if $sh(T) = j > 1$ then $||o'|| \leq O(hyp(w + 1, m, j - 1))$.

If the variables of Q are $x_1, \ldots, x_k$, it follows that the space needed to write a given instantiation of these variables is $O(hyp(w + 1, m, i - 1))$. To complete the proof of (**1**) it is easily verified that for each $w, i, c, c' > 0$ there is a $g \in H_{i-1}$ such that $c' \cdot hyp(w + 1, n + c, i - 1) \leq g(n)$ for each $n \geq 0$.

(2) Suppose now that $f \in$ QTIME(H_{i-1}) is a C-generic database mapping from D to T where D is a database schema such that the set-height of each type in D is 0 and T is a type with set-height 0. This implies the existence of a Turing machine M which tests $o \in f(d)$ for objects o of type T and input database instances d of D, and

has running time $\leq g$ for some g in H_{i-1}. The basic idea of the proof is to use the construction of Example 3.5 to build a query Q_M which uses a simulation of M. A major concern of the proof is the encoding of arbitrary inputs d, o (which use elements from the infinite set $\mathbf{U}$) into the finite tape alphabet of M.

We begin the technical development by choosing a type T_{big} which is "big enough" to serve as the indices for storing computations of M. Let m be the sum of the widths of the output schema and the relations in the input schema for f. Define the function $h(n) = (m+c)n(n^m+1)$. It is easily verified that $h(n) \geq (m+c)(n^m+1)\log n$ is an upper bound on the length of a possible input for M given an input database d and a possible output o, where $n = |adom(d) \cup C|$, c is the number of delimiters used to encode a tuple, and assuming that $adom(o) \subseteq adom(d, Q)$. Obviously, $g'(n) = g(h(n))$ is in H_{i-1}. Now choose $n_0 \geq 0$ and $w \geq 0$ such that $hyp(w, n, i-1) \geq g'(n)$ for each $n \geq n_0$. Let T_{big} be the "largest" type with set-height $i-1$ and branching w (i.e., T_{big} has a tuple-node root, each tuple-node has w children, each set-node has a tuple-node as its child, and each maximal branch of T_{big} has $i-1$ set-nodes). Thus, if the query Q_M uses at least n_0 constants, then on an input d with $|adom(d) \cup C| = n$, we will have $|adom(d, Q_M)| \geq n_0$ and so $|cons_{adom(d,Q_M)}(T_{\text{big}})| \geq g'(n)$.

Let $C = \{c_1, ..., c_r\}$. Without loss of generality, we assume that M uses constants in C explicitly, the distinguished symbols $0, 1$ which will be used for the encoding of other domain elements, and also other delimiters for encoding d into a sequence, i.e., the tape alphabet of M is $\Sigma \supseteq \{c_1, ..., c_r, 0, 1\}$.

Now the construction of the calculus query comes down to encoding the elements of $adom(d)$ into Σ^*; encoding the input database d and some object o of type T into a sequence for M; simulating the behavior of M on the encoded inputs; and interpreting the outputs of M. Since the c_i's are actually tape symbols they will not be encoded. Each other element appearing in the active domain of the input will be encoded by a sequence over $\{0, 1\}$, whose length is $\leq O(\lceil log(|adom(d)|)\rceil)$. In more detail, a variable v of type $\{[U, U, U, U]\}$ is used to hold the encoding where, for an element $e \in adom(d) - C$, v contains the following tuples:

e_1	e_2	e	$\sigma_{e,1}$
e_2	e_3	e	$\sigma_{e,2}$
	...		
e_q	e_{q+1}	e	$\sigma_{e,q}$

The chain in the first two columns is an initial segment of a total ordering of $adom(d, Q_M)$ which is provided by the query, and the last column holds a sequence of 0's and 1's. In detail, given a total order for $adom(d, Q_M)$ stored in a variable s, a formula $\xi_{\text{ENC}}(s, v)$ can be constructed in the calculus which states that v holds a relation such that for every element in the third column the first two columns hold an initial segment of s and for every pair of elements in the third column their encoding (the fourth column) are different, i.e., v holds encodings of all elements in the active domain of the input. It is further enforced in $\xi_{\text{ENC}}(s, v)$ that v holds such a relation

with the smallest length. This then ensures that the length of the encoding relation is $\leq O(n\lceil log(n)\rceil)$ where $n = |adom(d)|$.

Similar to Example 3.5, we use $T_M = \{[T_{\text{big}}, T_{\text{big}}, U, U]\}$ to encode a computation of M. Suppose x of type T_M holds an encoding of a computation of M, using a total order s' on $cons_{adom(d,Q_M)}(T_{\text{big}})$. It is straightforward to write a formula $(\phi_{\text{ENC}}(s, v, s', t, x))$ which states that the computation starts from the encoding of the input database and a possible output t of type T. The query is now defined as

$$Q_M = \{t/T \mid \exists s/\{[U,U]\}\, \exists v/\{[U,U,U,U]\}\, \exists s'/\{[T_{\text{big}}, T_{\text{big}}]\}\, \exists x/T_M\, \psi\}$$

where

$$\psi = \text{ORD}_U(s) \wedge \text{ORD}_{T_{\text{big}}}(s') \wedge \xi_{\text{ENC}}(s, v) \wedge \phi_{\text{ENC}}(s, v, s', t, x) \wedge \text{COMP}_{M,T_M}(s', x) \wedge \zeta$$

where ORD_U and $\text{ORD}_{T_{\text{big}}}$ give total orderings; and COMP_{M,T_M} characterizes a successful computation in the spirit of Example 3.5; and $\zeta = \bigwedge_{j=1}^{n_0} b_j \approx b_j$ for some set $B = \{b_1, \ldots, b_{n_0}\}$ of constants, which is to ensure $|adom(d, Q_M)| \geq n_0$. The running time of M on an input d, o (where $adom(o) \subseteq adom(d, Q)$) is $\leq g'(n) \leq hyp(w, n, i-1)$ $\leq |cons_{adom(d,Q_M)}(T_{\text{big}})|$. It is now easily seen that Q_M expresses f. □

Corollary 4.5: (1) $\text{QSPACE}(H_{i-2}) \subseteq \text{CALC}_{0,i} \subseteq \text{QSPACE}(H_{i-1})$;
(2) $\text{QTIME}(H_{i-1}) \subseteq \text{CALC}_{0,i} \subseteq \text{QTIME}(H_i)$. □

The above yields:

Corollary 4.6: $\mathcal{E} \equiv \bigcup_{i\geq 0} \text{CALC}_{0,i}$. □

We conclude this section with a brief comparison of the results reported here with those reported in [KV88], which were developed simultaneously and independently. A relatively minor difference between the investigations is that [KV88] uses the Logical Data Model (LDM) [KV84] as its underlying data model, rather than using complex objects as done here. More importantly, the present paper focuses primarily on *expressive power* and *complexity*, while [KV88] focuses primarily on *complexity* (there called 'data complexity') and *expression complexity* (see [Var82]).

There is considerable correspondence between the results here and the results of [KV88] on (data) complexity. LDM calculus queries [KV84] directed at an LDM schema use an auxiliary schema for computation in a manner similar to the intermediate types defined here. In [KV88] queries are categorized by the set-height of these auxiliary schemas, and also by the quantifier structure on variables of highest set-height of the (prenex normal form) of the formula used by the query. Results in [KV88] describe an exact correspondence between the complexity of queries and families of alternating Turing machines, which are constrained to run in a fixed hyper-exponential amount of time, and which have a bounded number of alternations of 'existential' and 'universal' states (which correspond to the number of quantifier alternations in the formula).

[KV88] also studies the impact of permitting intermediate types whose tuples have differing widths. Thus, [KV88] provides a much more complete picture of the complexity of queries involving the set construct than that given in Theorem 4.4. In view of Remark 3.6, the complexity results of [KV88] can easily be transformed into analogs which characterize the expressive power of classes of queries in terms of alternating Turing machines. Notably, the results of [KV88] can easily be extended to yield Corollary 4.6. Indeed, results of [KV88] helped to clarify our understanding of our original proof of Corollary 4.6, leading us to generalize our earlier lower bound results [HS88] to obtain the first containment of Theorem 4.4.

The results of [KV88] imply a hierarchy result which is related to, but subtly different than, the Hierarchy Theorem presented in the next section. The result of this paper states that $\text{CALC}_{0,i} \sqsubset \text{CALC}_{0,i+1}$ for each $i \geq 0$. To indicate the flavor of the result of [KV88], let $\text{CALC}_{0,\exists i}$ ($\text{CALC}_{0,\forall i}$) be the family of queries in $\text{CALC}_{0,i}$ whose formulas are in prenex form, in which all variables of set-height i are existentially (universally) quantified and appear at the left end of the quantifier list. (This corresponds to the class $\exists{-}power(1,i)$ ($\forall{-}power(1,i)$) in [KV88]). Results in [KV88] imply that for each $i \geq 0$, $\text{CALC}_{0,\exists i} \sqsubset \text{CALC}_{0,\exists(i+1)}$. and $\text{CALC}_{0,\forall i} \sqsubset \text{CALC}_{0,\forall(i+1)}$.

5 The $\text{CALC}_{0,i}$ Hierarchy

Theorem 4.4 implies that $\text{CALC}_{0,i} \sqsubset \text{CALC}_{0,i+2}$ (because [4] $\text{QSPACE}(H_i) \subset \text{QSPACE}(H_{i+1})$) for each $i \geq 1$. This implies that the $\text{CALC}_{0,i}$ hierarchy does not collapse. The main result in the section, Theorem 5.1, refines this result, stating that $\text{CALC}_{0,i} \sqsubset \text{CALC}_{0,i+1}$ for each $i \geq 0$. The proof uses a spectrum theorem of [Ben62].

We now state and prove the Hierarchy Theorem:

Theorem 5.1: (Hierarchy) For each $i \geq 0$, $\text{CALC}_{0,i} \sqsubset \text{CALC}_{0,i+1}$, i.e., there is a query $Q \in \text{CALC}_{0,i+1}$ such that no query in $\text{CALC}_{0,i}$ is equivalent to Q.

This theorem is proved by reducing it to a classical theorem on *spectra* developed by Bennett [Ben62]. We begin the development by introducing needed terminology and stating Bennett's theorem. We then present the formal reduction. The development is largely straightforward; the discussion is terse, and in places, informal.

Bennett's theorem on spectra focuses on the expressive power of a higher-order predicate calculus language very similar to the one used in the current paper, but with a number of minor differences. In particular, Bennett's logic

1. uses a countable number of basic types;
2. does not permit variables of tuple type;
3. uses the expression '$x(x_1, ..., x_n)$' to denote, intuitively, '$[x_1, ..., x_n] \in x$';

[4] '$\subset$' denotes proper set inclusion.

4. uses typed variables, rather than specifying the types of variables through restricted quantification; and

5. uses the classical logic notation of '(x)' for '$\forall x$' and '$(\phi\psi)$' for '$\phi \wedge \psi$'; and does not use the symbols '$\exists$' or '$\vee$'.

Our reduction to the spectra theorem addresses each of the differences (1) through (4). With regards to (5) we retain the logical notation already used elsewhere in the present paper.

To provide a technical basis for our discussion we introduce a slight variant, called the *b-calculus*, of our calculus which is equivalent to the language of Bennett. We assume the existence of an infinite set of (*many-sorted*) *basic type symbols* $B_1, B_2, \cdots$. A *b-formula* is like a formula from our calculus, except that:

1. in types, the many-sorted type symbols B_i are used instead of U;

2. relation names do not occur (intuitively, the basic type symbols serve as unary relation names);

3. all free variables are typed by an implicit typing function, which assigns to each variable a set type (possibly with tuple subtypes) or a basic type;

4. we permit construction of terms of the form '$[x_1, ..., x_n]$', where each x_i is of basic or set type (this term is of the natural tuple type);

5. we do not permit terms of the form $t.i$, where t is a term and i a positive integer; and

6. atomic formulas are of the form '$x \approx y$' and '$[x_1, ..., x_n] \in x$' (where the natural typing restrictions are obeyed).

We focus on finite many-sorted models of b-formulas. Suppose that basic types B_{i_1}, ..., B_{i_s} include all basic types of variables occurring in a b-formula ϕ. Let $D_1, ..., D_s$ be finite disjoint sets of (atomic) objects. Intuitively, sequences $[D_1, \ldots, D_s]$ serve as the *structures* in this logic. In particular, if α is an assignment of free variables in ϕ to (atomic or constructed) objects of the appropriate types, then the notion of $[D_1, ..., D_s]$ *satisfies* ϕ under α, denoted $[D_1, ..., D_s] \models \phi[\alpha]$, is defined in the natural manner, where variable x of type B_i ranges over elements of D_i.

In the current paper the focus is on the set-height of variable types in calculus formulas. The spectra theorem is stated in terms of a related and more refined notion, called the *order* of b-formulas. There are a number of differences between how 'set-height' and 'order' are defined, the most obvious of which is that a set-height of n corresponds roughly to an order of $2n$. Furthermore, the notion of order has different treatment for quantified and free variables.

Definition: The *order*, denoted $o(\cdot)$, of subformulas of a b-formula ϕ is defined inductively as follows (where $sh(x)$ denotes the set-height of the type of x):

1. for arbitrary variables y and z of the same type, $o(y \approx z) = 1$
2. $o([y_1, ..., y_n] \in z) = 2 \cdot sh(z) - 1$
3. $o(\forall y\ \psi) = \max\{2 \cdot sh(y), o(\psi)\}$
4. $o(\exists y\ \psi) = \max\{2 \cdot sh(y), o(\psi)\}$
5. $o(\neg\psi) = o(\psi)$
6. $o(\psi \vee \theta) = \max\{o(\psi), o(\theta)\}$
7. $o(\psi \wedge \theta) = \max\{o(\psi), o(\theta)\}$

Note that all formulas have order at least 1. Also, if a b-formula has odd order $2n-1$, then it has at least one free variable with set-height n, and all quantified variables have set-height $< n$. Suppose that ϕ has no free variables. Then its order is 1 iff it has no higher-order types; otherwise $o(\phi)$ is 2 times the maximum set-height of types occurring in ϕ. Finally, suppose that ϕ has a free variable z which is used only in atomic formulas using '$\approx$'. Then the maximal set-height of types in ϕ is at least $sh(z)$, but $o(\phi)$ could be $< 2sh(z) - 1$. (It is for this reason that Lemma 5.3 is needed.)

Definition: Let ϕ be a b-formula with free variables $z_1, ..., z_l$; and $B_{i_1}, ..., B_{i_s}$ a sequence of basic types containing all basic types of variables occurring in ϕ. The *spectrum* for ϕ (relative to this sequence), denoted R_ϕ is the set of s-vectors of natural numbers

$$R_\phi = \{\langle k_1, ..., k_s\rangle \mid \text{for some finite disjoint sets } D_1, ..., D_s \text{ and objects } o_1, ..., o_l, \\ [D_1, ..., D_s] \models \phi[z_1/o_1, ..., z_l/o_l], \text{ and } |D_i| = k_i \text{ for } i \in [1..s]\}$$

R is a spectrum of *order* n if $R = R_\phi$ for some b-formula ϕ with $o(\phi) \leq n$. For $n \geq 1$, $\mathcal{S}^n$ is the set of all spectra with order n.

Note in the above definition that, intuitively speaking, free variables in formulas are essentially viewed as being existentially quantified.

We can now state:

Theorem 5.2: (Spectra) [Ben62] For each[4] $n \geq 1$, $\mathcal{S}^n \subset \mathcal{S}^{n+2}$. □

The proof of the Hierarchy Theorem revolves around showing that queries in $\text{CALC}_{0,i}$ can be used to simulate spectra in $\mathcal{S}^{2i}$, and vice versa. In more detail, we show for $i \geq 1$ that $\text{CALC}_{0,i} \equiv \text{CALC}_{0,i+1}$ implies $\mathcal{S}^{2i} = \mathcal{S}^{2i+2}$. (We focus on showing that $\text{CALC}_{0,i} \sqsubset \text{CALC}_{0,i+1}$ for $i \geq 1$, because Proposition 3.9 already implies the result for $i = 0$.) To do this, we assume that $\text{CALC}_{0,i} \equiv \text{CALC}_{0,i+1}$, and that $R \in \mathcal{S}^{2i+2}$. From R we obtain a b-formula ϕ of order $2i + 2$. ϕ is transformed into a query $Q \in \text{CALC}_{0,i+1}$; a key technicality focuses on the removal of free variables from ϕ (Lemma 5.3). From the assumption, there is a query $Q' \in \text{CALC}_{0,i}$ with $Q' \equiv Q$. Q' may have constants; it is transformed into an (essentially) equivalent query $Q''' \in \text{CALC}_{0,i}$

without constants (Lemma 5.5). Finally, Q''' is transformed into a b-formula ϕ' with order $2i$ (Lemma 5.6).

We now begin the formal argument. Let $i \geq 1$ be fixed, and suppose that $\text{CALC}_{0,i} = \text{CALC}_{0,i+1}$. To show that $\mathcal{S}^{2i} = \mathcal{S}^{2i+2}$ in this case, suppose that $R \in \mathcal{S}^{2i+2}$. Thus $R = R_\phi$ for some b-formula ϕ with order $\leq 2i+2$ and some sequence $B_{i_1}, ..., B_{i_s}$ of basic types. The following lemma shows that we can assume without loss of generality that ϕ has no free variables.

Lemma 5.3: If $n \geq 1$ and ξ is a b-formula of order $\leq 2n$, then there is a b-formula τ of order $\leq 2n$ with no free variables such that $R_\tau = R_\xi$.

Proof: We assume without loss of generality that ξ is in prenex, disjunctive normal form, and (as a result) that the set of free variables in ξ is disjoint from the set of quantified variables. Let $\mathcal{Z}$ be the set of free variables in ξ. We partition $\mathcal{Z}$ into sets $\mathcal{Z}_{\text{short}}$ and $\mathcal{Z}_{\text{tall}}$, where $z \in \mathcal{Z}_{\text{short}}$ if $sh(z) \leq n$, and $z \in \mathcal{Z}_{\text{tall}}$ if $sh(z) > n$. Intuitively, the variables in $\mathcal{Z}_{\text{short}}$ can be quantified without changing the order of ξ. We shall see that the variables in $\mathcal{Z}_{\text{tall}}$ can occur only in atomic formulas involving '$\approx$'; and thus enforce simple cardinality constraints on the input domains.

For the formal development, let $\mathcal{Z}_{\text{short}} = \{z_1, ..., z_t\}$, and $\rho = \exists z_1 \cdots \exists z_t \xi$. Then ρ has order $2n$ and, from the definition of spectrum, $R_\rho = R_\xi$.

Suppose now that a free variable z occurs in at least one atomic subformula of the form '$[x_1, ..., x_l] \in z$' or '$[x_1, ..., z, ..., x_l] \in y$'; and that $sh(z) = j$. Because z occurs in a subformula involving '$\in$', $2n \geq o(\xi) \geq 2 \cdot sh(z) - 1 = 2j - 1$. Thus, $j \leq n$ and so $z \in \mathcal{Z}_{\text{short}}$. In particular, this implies that all free variables in ρ occur only in atomic formulas of the form $x \approx y$. Furthermore, in these formulas both x and y are in $\mathcal{Z}_{\text{tall}}$, and are both free.

Let

$$\mu = \nu_1 \vee \cdots \vee \nu_m$$

be the matrix of ρ (and ξ), and for each $j \in [1..m]$ let

$$\nu_j = \beta_j \wedge \gamma_j,$$

where γ_j is the conjunction of all atomic subformulas of ν_j which involve a free variable, and β_j is the conjunction of remaining atomic subformulas of γ_j. It follows that only bound variables occur in β_j and only free variables occur in γ_j.

We now argue that μ can be transformed into an equivalent formula by replacing each formula γ_j by a formula γ'_j which involves exclusively basic type variables. Recall first that in the definition of spectrum, free variables are, intuitively speaking, treated as existentially quantified. Let $j \in [1..m]$ be fixed, and let $\gamma = \gamma_j$. Without loss of generality, γ has no literals of the form $x \approx y$, i.e., it is a conjunction of literals of the form $x \not\approx y$. Let $T_1, \ldots, T_q$ be the types of variables in γ. Partition γ into $\gamma^1 \wedge \ldots \wedge \gamma^q$, where for each $p \in [1..q]$, γ^p is a conjunction of all literals involving variables of type

T_p; and let

$$\gamma^p = \bigwedge_{u=1}^{v_p} x_1^{p,u} \not\approx x_2^{p,u}.$$

For $\vec{a} = \langle a_1, \ldots, a_s \rangle$ and $\vec{b} = \langle b_1, \ldots, b_s \rangle$ in $\mathbf{N}^s$, write $\vec{a} \leq \vec{b}$ if $a_i \leq b_i$ for each $i \in [1..s]$. From the form of γ^p it follows that: if $\vec{a} \in R_{\gamma^p}$ (i.e., the spectrum of γ^p) and $\vec{a} \leq \vec{b}$ then $\vec{b} \in R_{\gamma^p}$. For each $p \in [1..m]$ let

$$Min(p) = \{\vec{a} \in \mathbf{N}^s \mid \vec{a} \text{ is a minimal element of } R_{\gamma^p} \text{ under } \leq\}$$

For $\vec{a} \in Min(p)$, let $\lambda_{\vec{a}}$ be a formula using only basic type variables which states that for each $l \in [1..s]$ there are at least a_l elements of type B_{i_l}. Finally set

$$\gamma'_j (= \gamma') = \bigwedge_{p=1}^{q} \left(\bigvee_{\vec{a} \in Min(p)} \lambda_{\vec{a}} \right)$$

It is easily verified that $R_{\gamma'_j} = R_{\gamma_j}$ for each $j \in [1..m]$.

For each $j \in [1..m]$ let $\nu'_j = \beta_j \wedge \gamma'_j$, and let

$$\tau = \exists y_1 \cdots \exists y_l \bigvee_{j=1}^{m} (\beta_j \wedge \gamma'_j)$$

where $\{y_1, ..., y_l\}$ is the set of variables used by the γ'_j's. It is easily verified that τ has the desired properties. □

In view of the above lemma, we assume henceforth that ϕ has no free variables. We now construct a query Q mapping from $< P_1 : U, \ldots, P_s : U >$ to U such that for all input instances $d =< P_1 : I_1, \ldots, P_s : I_s >$,

$$Q[d] \;=\; \begin{cases} \cup_{j=1}^{s} I_j & \text{if } \langle |I_1|, \ldots, |I_s| \rangle \in R (= R_\phi) \\ \emptyset & \text{otherwise} \end{cases}$$

To do this we set

$$Q = \{t/U \mid \psi \wedge (P_1(t) \vee \cdots \vee P_s(t))\},$$

where ψ is a (normal) formula obtained by making the following modifications to the b-formula ϕ:

1. If x is a variable of basic type B_{i_j} then quantifications of the form '$\forall x \cdots$' are replaced by '$\forall x/U\ (P_j(x) \rightarrow \cdots)$', and quantifications of the form '$\exists x \cdots$' are replaced by '$\exists x/U\ (P_j(x) \wedge \cdots)$'. Also, quantified higher-order variables of ϕ are given appropriate types in ψ and then restricted so that the basic types at the leaves come from the appropriate P_j's.

2. Subformulas of the form '$[x_1, ..., x_k] \in x$' are replaced by '$\exists x'/T\ (x'.1 \approx x_1 \wedge \cdots \wedge x'.k \approx x_k \wedge x' \in x)$', where T is chosen appropriately.

It is clear that the resulting Q is a query in $\text{CALC}_{0,i+1}$ with the desired semantics. Note that Q is $\emptyset$-generic.

By the assumption that $\text{CALC}_{0,i} = \text{CALC}_{0,i+1}$, there is a query $Q' = \{t/U \mid \psi'\} \in \text{CALC}_{0,i}$ such that $Q' \equiv Q$. Note that although Q' may involve constants, it is also $\emptyset$-generic.

Let Q' use k (atomic) constants, let $m = \max\{sk, s^2\}$, and let

$$R' = R - \{\langle i_1, ..., i_s\rangle \mid \sum_{j=1}^{s} i_j \leq m\}$$

In the balance of the proof we will show that R' is a spectrum of order $2i$. As a result of the following lemma of Bennett, this will imply that R is also a spectrum of order $2i$.

Lemma 5.4: (Finite Modification) [Ben62] If R' is a spectrum of order $n \geq 0$ and $R - R'$ is finite, then R is a spectrum of order n. □

Showing that R' is a spectrum of order $2i$ involves two steps. We first show that the formula ψ' can be transformed to have no constants, and to return $\emptyset$ on "small" inputs (Lemma 5.5). We then show how to transform the result into a b-formula ϕ' of order $2i$ with spectrum R' (Lemma 5.6).

Lemma 5.5: Let $Q' = \{t/U \mid \psi'(t)\} \in \text{CALC}_{0,i}$ be a $\emptyset$-generic query with constants $C = \{c_1, ..., c_k\}$ which maps from $S =< P_1 : U, \ldots, P_s : U >$; and let $m \geq k \geq 0$. Then there is a query $Q''' = \{t/U \mid \psi'''(t)\}$ with no constants such that for each input $d =< P_1 : I_1, \ldots, P_s : I_s >$,

$$Q'''[d] \;=\; \begin{cases} Q'[d] & \text{if } \sum_{j=1}^{s} |I_j| \geq m; \text{ and} \\ \emptyset & \text{if } \sum_{j=1}^{s} |I_j| < m. \end{cases} \qquad (1)$$

Proof: We first address the empty-output condition for "small" inputs. In particular, it is straightforward to build a formula α which states, intuitively, that "$\Sigma_{j=1}^{s}|I_j| \geq m$". Let $Q'' = \{t/U \mid \psi''(t)\}$ where $\psi''(t) = \psi'(t) \wedge \alpha$; this clearly satisfies (1).

We write $\psi''(c_1, ..., c_k; t)$ to indicate the presence of the constants C in ψ'', and letting $z_1, ..., z_k$ be new variables, we write $\psi''(z_1, ..., z_k; t)$ to denote the result of replacing c_i by z_i in ψ'' for $i \in [1..k]$. Also, let

$$\psi'''(t) = \exists z_1 \cdots \exists z_k (\bigwedge_{i \neq j} z_i \not\approx z_j \;\wedge\; \psi''(z_1, ..., z_k; t))$$

and $Q''' = \{t/U \mid \psi'''(t)\}$. It now remains to show that $Q'''[d] = Q''[d]$ on all inputs d.

Because Q' is $\emptyset$-generic, Q'' is. Also, Q''' is $\emptyset$-generic because it involves no constants. In particular, then, for each input database d, atomic object c, and permutation π on $dom(U)$,

$$d \models_{adom(d)\cup C} \psi''[t/c] \;\Leftrightarrow\; \pi(d) \models_{adom(\pi(d))\cup C} \psi''[t/\pi(c)] \tag{2}$$

and

$$d \models_{adom(d)} \psi'''[t/c] \;\Leftrightarrow\; \pi(d) \models_{adom(\pi(d))} \psi'''[t/\pi(c)] \tag{3}$$

Now let $d =< P_1 : I_1, \ldots, P_s : I_s >$ be a fixed input instance and c a fixed atomic object. To conclude the proof it suffices to show that for each d there is some π such that

$$d \models_{adom(d)\cup C} \psi''[t/c] \;\Leftrightarrow\; \pi(d) \models_{adom(\pi(d))} \psi'''[t/\pi(c)] \tag{4}$$

We begin with the necessity direction of (4). In particular, suppose that $d \models_{adom(d)\cup C} \psi''[t/c]$. Because of the subformula α, $\Sigma_{j=1}^{s}|I_j| \geq m \geq sk$. This implies that $|adom(d)| \geq k$ (even if some of the I_j's have nonempty intersection). Thus, there is a permutation π of $dom(U)$ such that $C \subseteq adom(\pi(d))$. By (2) we have $\pi(d) \models_{adom(\pi(d))} \psi''[t/\pi(c)]$. It follows that

$$\pi(d) \models_{adom(\pi(d))} \exists z_1 \cdots \exists z_k (\bigwedge_{i\neq j} z_i \not\approx z_j \;\wedge\; \psi''(z_1, ..., z_k; t))[t/\pi(c)],$$

i.e., $\pi(d) \models_{adom(\pi(d))} \psi'''[t/\pi(c)]$. (3) now yields $d \models_{adom(d)} \psi'''[t/c]$ as desired.

For the sufficiency of (4), suppose that $d \models_{adom(d)} \psi'''[t/c]$, i.e., that

$$d \models_{adom(d)} \exists z_1 \cdots \exists z_k (\bigwedge_{i\neq j} z_i \not\approx z_j \;\wedge\; \psi''(z_1, ..., z_k; t))[t/c].$$

This implies that there are distinct atomic objects $c'_1, ..., c'_k$ such that

$$d \models_{adom(d)} (\bigwedge_{i\neq j} z_i \not\approx z_j \wedge \psi''(z_1, ..., z_k; t))[z_1/c'_1, ..., z_k/c'_k, t/c].$$

Choose a permutation π such that $\pi(c'_j) = c_j$ for $j \in [1..k]$. Then (a variant of) (3) implies

$$\pi(d) \models_{adom(\pi(d))} \exists z_1 \cdots \exists z_k (\bigwedge_{i\neq j} z_i \not\approx z_j \;\wedge\; \psi''(z_1, ..., z_k; t))[z_1/c_1, ..., z_k/c_k, t/\pi(c)],$$

and in particular,

$$\pi(d) \models_{adom(\pi(d))} \psi''(z_1, ..., z_k; t)[z_1/c_1, ..., z_k/c_k, t/\pi(c)],$$

i.e., $\pi(d) \models_{adom(\pi(d))} \psi''(c_1, ..., c_k; t)[t/\pi(c)]$. It now follows from (2) that $d \models_{adom(d)\cup C} \psi''[t/c]$ as desired. Thus (4) is demonstrated, and more generally, $Q''' \equiv Q''$. $\square$

The next lemma performs the transformation of $Q''' = \{t/U \mid \psi'''(t)\}$ into the b-formula ϕ'. Intuitively, the main problem here is that the b-formula is constrained to

use s different atomic types, while the formula ψ''' can use the members of the input I_j's interchangeably. To illustrate this concretely, suppose that $s \geq 3$, and consider the formula $\exists z/T\beta(z)$ where $T = \{[U, U]\}$ and β says, intuitively, that z holds a 1-1 correspondence witnessing that $|I_1| + |I_2| = |I_3|$. Suppose in particular that β does this by using the first coordinate of z-elements to hold elements of $I_1 \cup I_2$ and the second coordinate of z-elements to hold elements of I_3. In this case β cannot be transformed in the naive fashion into a b-formula, because in a b-formula the type of z must use either D_{i_1} or D_{i_2} for the first coordinate, not their union. To circumvent this problem, the b-formula we ultimately build will use the largest of the input domains to simulate all of the input domains.

Lemma 5.6: Let $Q''' = \{t/U \mid \psi'''(t)\} \in \mathrm{CALC}_{0,i}$ have no constants and input schema $S =< P_1 : U, \ldots, P_s : U >$; and let $m \geq s^2$. Suppose further that for each input $d =< P_1 : I_1, \ldots, P_s : I_s >$

1. $Q'''[d] = \emptyset$ or $Q'''[d] = \cup_{j=1}^{s} I_j$; and
2. $Q'''[d] = \emptyset$, if $\Sigma_{j=1}^{s} |I_j| < m$.

Then there is a b-formula ϕ' of order $\leq 2i$ such that

$$R_{\phi'} = \{\langle |I_1|, \ldots, |I_s| \rangle \mid Q'''[\langle P_1 : I_1, ..., P_s : I_s \rangle] = \cup_{j=1}^{s} I_j\}$$

Proof: Recall that the input to the b-formula ϕ' is an s-tuple $[D_1, ..., D_s]$. Intuitively, the formula ϕ' will "execute" three distinct phases of "activity":

1. find the first j such that $|D_j| = \max\{|D_i| \mid i \in [1..s]\}$.
2. using that j, develop an encoding of elements of D_i, $i \in [1..s]$, which uses only elements of D_j.
3. simulate the behavior of ψ''' using the encoded objects.

We discuss each of these phases in turn.

For the first phase, because ϕ' is to have order $2i \geq 2$, it is straightforward to obtain subformulas γ_j, $j \in [1..s]$, where

$$[D_1, ..., D_s] \models \gamma_j \;\Leftrightarrow\; j \text{ is least such that } |D_j| = \max\{|D_i| \mid i \in [1..s]\}$$

Note that on a given input, exactly one of the γ_j's will be true. Also, note that if γ_j is true, then by the choice of m, $|D_j| \geq s$. The ultimate formula ϕ' we build will have the form

$$\phi' \;=\; (\gamma_1 \wedge \phi'_1) \vee \cdots \vee (\gamma_s \wedge \phi'_s).$$

We now describe the formula ϕ'_j, which accomplishes phases (2) and (3) above, using the coordinate j. Phase (2) is the development of an encoding of $\cup_{i=1}^{s} D_i$ into the type $[D_j, D_j]$. In particular, ϕ'_j has the form

$$\exists f_1 \cdots \exists f_s \exists e_1 \cdots \exists e_s (\bigwedge_{i \neq i'} f_i \not\approx f_{i'} \;\wedge\; \bigwedge_{i=1}^{s} \epsilon_i(e_i) \;\wedge\; \widehat{\phi_j}))$$

where

1. f_i has type D_j for $i \in [1..s]$;
2. e_i has type $\{[D_i, D_j, D_j]\}$ for $i \in [1..s]$;
3. ϵ_i says, intuitively, that e_i holds a 1-1 into map of D_i to $\{f_i\} \times D_j$, for $i \in [1..s]$; and
4. $\widehat{\phi_j}$ is a formula described shortly which accomplishes phase 3.

For the remainder of the proof we assume that b-formulas can contain variables x with type $[D_j, D_j]$, and that terms $x.1$ and $x.2$ can be used. We also assume that the type $[D_j, D_j]$ can occur as a subtype in the types of b-formula variables. These assumptions will make the exposition intuitively more clear, and transforming the construction we describe into actual b-formulas is straightforward.

For a (normal) type T, define $\widehat{T}$ to be the b-calculus "type" obtained by replacing each leaf of T by $[D_j, D_j]$. We also make use of special formulas $\delta_T(\hat{x})$ where

1. T is a normal type;
2. $\hat{x}$ is of type $\widehat{T}$; and
3. $\delta_T(\hat{x})$ says, intuitively, that $\hat{x}$ is built out of elements of $\pi_{2,3}(\cup_{i=1}^{s} e_i)$, i.e., that each underlying atomic pair in $\hat{x}$ is the encoding of one of the original input objects.

Note that if x is of basic type, then $\hat{x}$ is of type $[D_j, D_j]$.

We now describe phase (3), which is accomplished by the formula(s) $\widehat{\phi_j}$ introduced as part of the formula ϕ'_j. In particular,

$$\widehat{\phi_j} = \exists \hat{t}(\delta_U(\hat{t}\,) \wedge \widehat{\psi'''}(\hat{t}\,)),$$

where $\widehat{\psi'''}(\hat{t}\,)$ is obtained from $\psi'''(t)$ by making two stages of modifications. First, a hybrid formula $\chi(t)$ is constructed from $\psi'''(t)$ by performing the following recursive modifications to eliminate all variables of tuple type.

1. Replace '$\forall x/[T_1, ..., T_k]\theta$' by '$\forall x_1/T_1 \cdots \forall x_k/T_k\theta'$', where θ' is constructed from θ by replacing expressions of the form
   (a) '$x.i$' by 'x_i'; and
   (b) '$x \in y$' by '$[x_1, ..., x_k] \in y$'
2. Replace '$\exists x/[T_1, ..., T_k]\theta$' by '$\exists x_1/T_1 \cdots \exists x_k/T_k\theta'$', where θ' is constructed from θ as in the '$\forall$' case.

Note that all variables in $\chi(t)$ are of set or basic type. We now construct $\psi'''(\hat{t}\,)$ from $\chi(t)$ using the following modifications:

3. Replace each variable x of type T by a variable $\hat{x}$ of type $\widehat{T}$.

4. Replace '$\forall x/T \cdots$' by '$\forall \hat{x}(\delta_T(\hat{x}) \rightarrow \cdots)$'.

5. Replace '$\exists x/T \cdots$' by '$\exists \hat{x}(\delta_T(\hat{x}) \wedge \cdots)$'.

6. Replace '$P_i(x)$' by '$\hat{x}.1 \approx f_i$'.

It is now straightforward to verify that $R_{\phi'}$ is the desired spectrum. □

By Lemma 5.6 we have shown that $R_{\phi'} = R'$ is a spectrum of order $2i$. By Lemma 5.4, R is also a spectrum of order $2i$. Since R was an arbitrary element of $\mathcal{S}^{2i+2}$, this implies that $\mathcal{S}^{2i} = \mathcal{S}^{2i+2}$, a contradiction. This completes the proof of Theorem 5.1.

6 Invented Values

As illustrated by Examples 3.5 and 3.7, complex object queries can use intermediate types to provide large sets for encodings. In this section we experiment with several alternative semantics for queries in an attempt to isolate this "use" of intermediate types from other uses. The semantics to be introduced are all based on *invented values*, i.e., atomic values not occurring in the database nor in the query (see also [AV87]). Some of these restrict the number of invented values used and hence generalize the notions of the *limited* and *unlimited* interpretations. For instance, countable invention, which allows a countably infinite set of invented values, is essentially the unlimited interpretation. These kinds of semantics allow us to explore queries which are not domain independent (or safe).

We now describe the main results presented in this section. First, it is shown that under the semantics of *finite* and *countable invention*, $\mathrm{CALC}_{0,1}$ subsumes $\bigcup_{0 \leq i} \mathrm{CALC}_{0,i}$, i.e., the hierarchy collapses at level one, in contrast with Theorem 5.1. Furthermore, invented values permit a "universal type" T_{univ} in τ_1 which can simulate any type from $\bigcup_{0 \leq i} \tau_i$. Motivated by the use of this universal type, we then discuss briefly *bounded invention* in which the number of invented values to be used is explicitly specified. Proposition 6.10 provides a correspondence between levels of nested sets and hyper-exponential quantities of invented values.

We also study the expressive power of calculus queries under the semantics of finite and countable invention. It turns out that finite invention has strictly richer expressive power than $\mathcal{E}$ (which uses the limited interpretation, i.e., no invention) but strictly weaker than countable invention. Finite invention allows queries which are not recursive and countable invention allows queries which are not recursively enumerable.

As an aside, we note that when restricted to the relational calculus, the families of queries under the semantics of finite and countable invention are expressively equivalent to that under the limited interpretation. In other words, the relational calculus with the unlimited interpretation has the same expressive power as with the limited interpretation. A key portion of this result was demonstrated in [AGSS86], and the

complete result was independently developed in [HS88] (details appear in [HS89a]). Also, our study of countable invention is extended in [HS89c], where it is shown to have essentially the same expressive power as the arithmetic hierarchy.

Finally, we discuss the relationship between the calculus with invention and "computable queries" studied by Chandra-Harel, Abiteboul-Vianu etc. One main difference is that computable queries are generally partial while calculus queries here are all totally defined. The other is that calculus queries with either finite or countable invention may not be computable. Hence, a notion of "terminal invention" is proposed and its semantics is shown to be essentially equivalent to the computationally complete languages **QL** [CH80] and **detTL** [AV88].

For the formal development, we view each class $\text{CALC}_{k,i}$ to be a set of purely *syntactic* objects. Recall from Section 2 that for a query $Q = \{t/T \mid \phi(t)\}$, a set $Y \subseteq \mathbf{U}$, and an input instance d, $Q|^Y[d] = \{o \mid d \models_Y \phi[t/o]\}$. We now provide the following definitions.

Definition: Let query $Q = \{t/T \mid \phi(t)\} \in \text{CALC}_{k,i}$ map from D to T, d be an instance of D, and $Y \subseteq \mathbf{U}$. Then

$$Q|_Y[d] \quad = \quad Q|^Y[d] \bigcap cons_{_{adom(d,Q)}}(T).$$

Further, if $|Y - adom(d,Q)| = n$ ($n \in \mathbf{N}$ or $n = \omega$), we define: $Q|_n[d] = Q|_Y[d]$.

The semantics of $Q|_n[d]$ is well defined since atomic values outside of the active domain are "generic" [Hul86] and thus indistinguishable as far as query evaluation is concerned. This is stated formally in the following proposition (proof omitted).

Proposition 6.1: Let Q be a query from D to T and d a database instance of D. For any subsets X, Y of $\mathbf{U}$, if $|X - adom(d,Q)| = |Y - adom(d,Q)|$, then: $Q|_X[d] = Q|_Y[d]$. □

Definition: Let Q be a query expression in $\text{CALC}_{k,i}$ and d a database, the value of Q on d using the semantics of *finite invention* is defined by

$$Q^{fi}[d] = \bigcup_{0 \le n < \omega} Q|_n[d]$$

and the value of Q using the semantics of *countable invention* is defined as

$$Q^{ci}[d] = Q|_\omega[d].$$

In the remainder of the section, we use Q^{ni} to denote the mapping defined by Q under the limited interpretation ('ni' stands for 'no invention'). We also define:

- $\text{CALC}^{ni}_{k,i} = \{\ Q^{ni} \mid Q \in\ \text{CALC}_{k,i}\}$;
- $\text{CALC}^{fi}_{k,i} = \{\ Q^{fi} \mid Q \in\ \text{CALC}_{k,i}\}$; and
- $\text{CALC}^{ci}_{k,i} = \{\ Q^{ci} \mid Q \in\ \text{CALC}_{k,i}\}$.

In the definition of finite invention, the answer is formed from a countable union of "queries", each of which uses a *finite* number of invented values. On the other hand, under the semantics of countable invention, any quantified variable x of type T can be viewed as ranging over $cons_{\mathrm{U}}(T)$ (a countably infinite set) without other restrictions. Recall that all constructed objects have finite active domain. In particular, the answer to a query under either form of invention is finite.

The following two examples demonstrate how invention yields richer expressiveness. Speaking intuitively, invention allows query processors to use internal values (transparent to users) during evaluation. As we shall see, these internal values provide some of the functionality of sets.

Example 6.2: Recall Example 3.7. Consider the mapping $f : D \to T$ defined so that $f(I) = I$ if there are numbers $p, q \geq 1$ such that $p^4 + 1 = q^7$, and $f_0(I) = \emptyset$ otherwise. There is a query in $\mathrm{CALC}_{0,1}^{fi}$ which expresses this mapping.

Consider now the functions f_i and queries Q_i from Example 3.7. In $\mathrm{CALC}_{0,1}^{fi}$ it is possible to inductively describe a sequence of variables x_j, $0 \leq j \leq i$, of type $\{U\}$ such that (intuitively) $|x_j| = hyp(1, |I|, j)$. Using this, it can be shown that for each i there is a query $Q_i' \in \mathrm{CALC}_{0,1}^{fi}$ such that $Q_i^{ni} \equiv Q_i'^{fi}$. □

Example 6.3: Recall Example 3.5, in which the type $\{[T, T, U, U]\}$ is used to hold the encoding of a Turing machine computation, where T is a type of set-height i, and $cons_A(T)$ serves as the index set for the different moves and tape positions of the simulation. Let M be a Turing machine which computes a query Q and which always halts within hyper-exponential time. Under finite invention, it is possible to replace T by U and to store the computation of M in a variable of type $\{[U, U, U, U]\}$. It follows that M can be simulated by a query using intermediate types of set-height 1, and so the query Q can be expressed in $\mathrm{CALC}_{0,1}^{fi}$. Thus, $\mathrm{CALC}_{0,i}^{ni} \sqsubset \mathrm{CALC}_{0,1}^{fi}$. (See also Lemma 6.15.) □

We now show that the $\mathrm{CALC}_{0,i}^{fi}$ and $\mathrm{CALC}_{0,i}^{ci}$ hierarchies collapse. In fact, we show something stronger, namely that there is a "universal type" in τ_1 which can serve as the sole intermediate type for the simulations needed to show these results.

Theorem 6.4: $\bigcup_{0 \leq i} \mathrm{CALC}_{0,i}^{fi} \equiv \mathrm{CALC}_{0,1}^{fi}$ and $\bigcup_{0 \leq i} \mathrm{CALC}_{0,i}^{ci} \equiv \mathrm{CALC}_{0,1}^{ci}$.

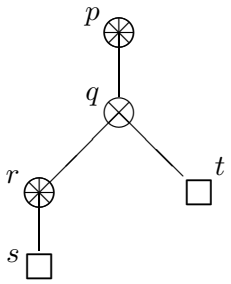
The above theorem follows immediately from the next lemma.

Lemma 6.5: Let Q be any query expression in $\mathrm{CALC}_{0,i}$. Then:

(a) there is a $Q' \in \mathrm{CALC}_{0,1}$ such that $Q'^{fi} \equiv Q^{fi}$; and

(b) there is a $Q'' \in \mathrm{CALC}_{0,1}$ such that $Q''^{ci} \equiv Q^{ci}$.

Furthermore, Q' and Q'' can be chosen to have exactly one intermediate type T_{univ} in τ_1.



(a) type T

$\{[\{a, b\}, c], [\emptyset, b]\}$

(b) object o of T

s,t	
i1	a
i2	b
i3	c

r	
i4	$\emptyset$
i5	{i1, i2}

p	
i8	{i6, i7}

q	
i6	[i4, i2]
i7	[i5, i3]

(c) intermediate encoding of o

[	p,	i8,	0,	i6]
[	p,	i8,	0,	i7]
[	q,	i6,	1,	i4]
[	q,	i6,	2,	i2]
[	q,	i7,	1,	i5]
[	q,	i7,	2,	i3]
[	r,	i4,	0,	0]
[	r,	i5,	0,	i1]
[	r,	i5,	0,	i2]
[	s,	i1,	0,	a]
[	s,	i2,	0,	b]
[	t,	i1,	0,	a]
[	t,	i2,	0,	b]
[	t,	i3,	0,	c]

(d) encoding of o in type T_{univ}

Figure 3: Illustration of encoding using the universal type T_{univ}

The proof of this lemma is given after defining the type T_{univ} and presenting a motivating example.

Notation: $T_{univ} = \{[U, U, U, U]\}$.

The main idea of Lemma 6.5 is to encode objects using invented values. The following example illustrates how T_{univ} can be used to encode an object of arbitrary type. The encoding is closely related to the representation of data in the LDM.

Example 6.6: Suppose that Q is a query expression in $\text{CALC}_{0,i}$, and that we are building a query $Q' \in \text{CALC}_{0,1}$ such that $Q'^{fi} \equiv Q^{fi}$. Suppose further that the type T of Figure 3(a) is used as an intermediate type by Q. In Q' we will simulate T and its constructive domain using T_{univ} and its constructive domain.

For the simulation we assume that the nodes of T are labeled, and use two sets of constants:

- the *node identifiers* of T, i.e., $\{p, q, r, s, t\}$; and
- the *coordinates* which can arise in T, in this case {0, 1, 2}, where 0 is used for objects associated with non-tuple nodes.

We also use a set of invented values which will serve as

- *object identifiers*, which we denote here as $\{i1, i2, i3, \ldots\}$. (The number of these depends on the object to be encoded.)

Suppose now that o is the object of type T shown in Figure 3(b). As an aid in understanding our encoding, we first develop an intermediate representation of o in the spirit of the LDM. As shown in Figure 3(c), for each distinct subtype of T we have a table which associates unique identifiers to values. Finally, the tables of this intermediate encoding are encoded into T_{univ} as follows:

- atom: encoded as a tuple [*node*, *id*, 0, *value*];
- tuple: encoded as a set of tuples [*node*, *id*, *coordinate*, *value*]; and
- set: encoded as a set of tuples [*node*, *id*, 0, *value*] ($\emptyset$ is encoded as [*node*, *id*, 0, 0]).

Figure 3(d) shows the final encoding of o into an object of type T_{univ}. □

Proof of Lemma 6.5: First, we show that for any query $Q \in \text{CALC}_{0,i}$, we can construct a $Q' \in \text{CALC}_{0,1}$, such that $Q^{fi} \equiv Q'^{fi}$; and further that the only intermediate types in τ_1 of Q' are T_{univ} and $\{U\}$. It is clear that Q' can be further transformed to have T_{univ} as the only intermediate type. The construction consists of two parts: (a) encode objects of intermediate types into objects of type T_{univ}; and (b) modify the formula of Q.

Suppose now that $Q = \{t/T_0 \mid \psi_0(t)\}$ and v of type $\{U\}$ is a new variable not appearing in ψ_0. Intuitively, the variable v holds invented values. Let $\delta_{_T}(x)$ ($\delta^v_{_T}(x)$) be a formula representing that x of type T is built from only elements in the active domain of input database and Q (and v). Now let

$$\psi_1(t) = \exists v/\{U\}\,(\forall x/U\,(x \in v \leftrightarrow \neg\delta_{_U}(x)) \wedge \theta(t, v))$$

where $\theta(t, v)$ is a formula obtained from ψ_0 according the following recursive transformation:

1. Replace '$\forall x/T \cdots$' by '$\forall x/T\,(\delta^v_{_T}(x) \rightarrow \cdots)$'; and
2. Replace '$\exists x/T \cdots$' by '$\exists x/T\,(\delta^v_{_T}(x) \wedge \cdots)$'.

It is obvious that Q and $Q_1 = \{t/T \mid \psi_1(t)\}$ are equivalent under finite invention. In the following discussion, we assume that w is also a new variable of type $\{U\}$.

The encoding of objects in part (a) is done in the way discussed in the Example 6.6. Suppose that T is an intermediate type of Q. Note that for a given object o' of type T_{univ}, and a set X of atomic values, we can check in the calculus whether o' is the encoding of an object in $cons_{_X}(T)$. Also, the equality among objects of type T becomes equality with isomorphism on object identifiers among objects of type T_{univ}. Finally, we can write a formula $\phi_{_T}(x, v, w)$ which states that x holds an encoding of an

object of type T, all atoms of x are in v or the active domain (of input), and all object identifiers are in w.

We pause to give some of the intuition behind our construction. Basically, the query Q' under finite invention (and using only types $U, \{U\}$, and T_{univ}) must simulate $Q_1|_X[d]$, where X is some set of invented values. A problem here is to distinguish the invented values in X from the invented values used to serve as object identifiers when encoding objects into T_{univ}. In the construction, we use the variable v to hold the elements of X, and the variable w to hold a disjoint set of invented elements to serve as object identifiers. A second problem is to ensure that the set held by w is "big enough" to perform the simulation. Both issues are addressed below.

We now focus on the construction of the query formula of Q' from $\psi_1(t)$. The main idea here is to encode all input relations; modify $\theta(t, v)$ recursively while preserving the semantics; and then decode objects of T_{univ} into output. To do that, we first describe a family of formulas, each of these incorporates the condition that $\phi_{_T}(x', v, w)$ ($\phi_{_T}(y', v, w)$ if applicable) for the appropriate type T. The formulas are (details omitted):

1. $\phi_{enc,T}(x, x', v, w)$ (where $T \in \tau_0$) means that x' is the encoding (relative to T) into T_{univ} of object x of type T with active domain contained in $adom(d, Q) \cup v$ and all object identifiers used are from w;

2. $\phi_{_R}(x', v, w)$ (for input predicate R) means that x' encodes an object x satisfying $R(x)$, and all object identifiers used are from w (i.e., $\phi_{_R}(x', v, w) = \exists x/T\,(R(x) \wedge \phi_{enc,T}(x, x', v, w))$, where T is the type of R);

3. $\phi_{\approx,T}(x', y', v, w)$ means the object encoded by x' (relative to T) = the object encoded by y' (relative to T) and all object identifiers used are from w;

4. $\phi_{\in,S,T}(x', y', v, w)$ means the object encoded by x' (relative to S) is a member of the object encoded by y' (relative to T) and all object identifiers used are from w; and

5. $\phi_{\in v}(x', v, w)$ means the object encoded by x' is an element of v and the object identifier used is from w (i.e., $x' = \{[p, i, 0, a]\}$ where $a \in v$ and $i \in w$).

Using these formulas, we transform $\theta(t, v)$ into $\theta'(t', v, w)$ by recursively substituting:

- $R(x)$ by $\phi_{_R}(x', v, w)$;
- $(x_1 \approx x_2)$ by $\phi_{\approx,T}(x'_1, x'_2, v, w)$ if x_1, x_2 are of type T;
- $(x_1 \in x_2)$ by $\phi_{\in,S,T}(x'_1, x'_2, v, w)$ if x_1, x_2 are of type $S, \{S\}$ respectively and $x_2 \neq v$;
- $(x \in v)$ by $\phi_{\in v}(x', v, w)$;
- $\forall x/T\,\psi(x)$ by $\forall x'/T_{univ}\,(\phi_{_T}(x', v, w) \rightarrow \psi'(x', v, w))$; and

- $\exists x/T\,\psi(x)$ by $\exists x'/T_{univ}\,(\phi_{_T}(x', v, w) \wedge \psi'(x', v, w))$.

Notice that in order for the above transformation to work, w must be disjoint from the active domain and v, and contain enough elements for all object identifiers needed. Fortunately, for the fixed query Q, the maximum set-height (of its types) is fixed. Hence the number of necessary identifiers is determined by a fixed hyper-exponential function $s_{_Q}$ and the size of the active domain (of the input and Q) or v, whichever is bigger. So we can write a formula $\beta(v, w)$ (whose only intermediate types in τ_1 are T_{univ} and $\{U\}$) which states that the size of w is sufficiently large. Now Q' is defined as $\{t/T_0 \mid \psi_1'(t)\}$ where

$$\psi_1'(t) = \exists w/\{U\}\,\exists v/\{U\}\,(\beta(v, w) \wedge \gamma(v, w) \wedge \theta''(t, v, w))$$

and[5]

1. $\gamma(v, w) = \forall x/U\,(x \in w \oplus x \in v \oplus \delta_{_U}(x))$; and
2. $\theta''(t, v, w) = \exists t'/T_{univ}\,(\phi_{_{enc,T_0}}(t, t', v, w) \wedge \theta'(t', v, w))$.

To show $Q^{fi} \equiv Q'^{fi}$, we need to prove that $Q^{fi}[d] = Q'^{fi}[d]$ for each database instance d. By the definition of finite invention, it is sufficient to show

(i) for each n, $Q|_n[d] \subseteq \cup_{0\leq m} Q'|_m[d]$; and

(ii) for each m, $Q'|_m[d] \subseteq \cup_{0\leq n} Q|_n[d]$.

Suppose d is a database instance, X and Y are arbitrary finite subsets of $\mathbf{U}$ such that X, Y, and $adom(d, Q)$ are pairwise disjoint, $d \models_{_{X\cup Y}} \beta(v, w)[v/X, w/Y]$, and o is an object $cons_{_{adom(d,Q)}}(T_0)$. It can be shown by induction on formulas that:

$$d \models_{_X} \theta(t, v)[t/o, v/X] \iff d \models_{_{X\cup Y}} \theta''(t, v, w)[t/o, v/X, w/Y]. \tag{5}$$

To show (i), let an object $o \in Q|_n[d]$. Hence $o \in Q_1|_n[d]$. By definition, $d \models_{_X} \theta(t, v)[t/o, v/X]$ for some X with $X \cap adom(d, Q) = \emptyset$ and $|X| = n$. Choosing a sufficiently large m, i.e., m such that there exist Y disjoint from X and $adom(d, Q)$ with $|X \cup Y| = m$ and $d \models_{_{X\cup Y}} \beta(v, w)[v/X, w/Y]$, we have immediately $o \in Q'|_m[d]$ by (5) above. For (ii), suppose now $o \in Q'|_m[d]$. Then $d \models_{_{X\cup Y}} \theta''(t, v, w)[t/o, v/X, w/Y]$. Obviously o is in $Q_1|_n[d]$ thus in $Q|_n[d]$ for $n = |X|$. Hence, $Q^{fi} \equiv Q'^{fi}$.

The argument for part (b) of the lemma is almost identical to the argument for part (a) just given. The key difference is that for part (b), we cannot use explicit variables v and w, because they would need to "hold" countably infinite sets. Actually, v and w were introduced into the argument (a) primarily for pedagogical reasons — in the encodings of objects x by x' it is not necessary to assume that the original invented values and the object identifiers are taken from disjoint sets. With this in mind, it is now straightforward to modify the argument for (a) to demonstrate (b). □

[5] $\oplus$ is "exclusive or".

More generally, for each $k \geq 0$ and $i \geq 1$, since $\text{CALC}_{k,i}$ allows the use of the universal type T_{univ}, $\text{CALC}^{fi}_{k,i}$ and $\text{CALC}^{ci}_{k,i}$ collapse. In particular, we have the following generalization of Theorem 6.4:

Theorem 6.7: If $k \geq 1$, then $\bigcup_{0 \leq i} \text{CALC}^{fi}_{k,i} \equiv \text{CALC}^{fi}_{k,max(1,k-1)}$ and $\bigcup_{0 \leq i} \text{CALC}^{ci}_{k,i} \equiv \text{CALC}^{ci}_{k,max(1,k-1)}$. □

The above result cannot be strengthened to state that $\bigcup_{0 \leq i} \text{CALC}^{fi}_{k,i} \equiv \text{CALC}^{fi}_{k,1}$ for $k > 2$. This is because transforming an input variable x of type T with $sh(T) = k$ will require variables of set-heights $k-1, \ldots, 1$. The analogous remark holds for countable invention.

Remark 6.8: The statement of Lemma 6.5 can be refined to use $T'_{univ} = \{[U, U]\}$, which is the smallest possible. It is interesting to compare this refinement of Lemma 6.5 with discussions in the folklore of semantic data modeling stating, intuitively, that any semantic model can be "reduced" to a Binary Data Model (such as [Abr74]), whose only data structure, essentially, is binary relation. Intuitively, the result here indicates two approaches to measuring the expense of such reductions: considering either (a) the number of additional invented values needed to perform the simulations, or (b) the size of the encoding of the relevant instance. The measure of case (a) focuses on the set-height of types in the original database. The measure of case (b) focuses on the space needed to write the (encoded) database (using a naive representation). Note that the space needed to write the encoded database will in some cases be considerably smaller than that needed to write the original database. □

On the other hand, if the set construct is extended to include arbitrary finite and infinite sets instead of only finite sets, the collapsing results for countable invention in Theorem 6.5 and Theorem 6.7 are false as shown in the following proposition (see appendix for the proof).

Proposition 6.9: If objects built using infinite sets are permitted, then for each $i \geq 0$, $\text{CALC}^{ci}_{0,i} \sqsubset \text{CALC}^{ci}_{0,i+1}$. □

As we saw above, the set construct in intermediate types is "equivalent" to having some invented values when considering expressive power. In the following, we discuss another kind of invention, *bounded invention*, which captures an even closer relationship between "sets" and invented values. By bounded invention, we mean the number of invented objects is bounded by a function based on the size of the input:

Definition: Let f be a function from $\mathbf{N}$ to $\mathbf{N}$. For a query $Q \in \text{CALC}_{k,i}$ and a database instance d for Q, the value of Q under *bounded invention* with f is defined as:

$$Q|_f[d] = \cup\{Q|_n[d] \mid n \leq f(|adom(d)|)\}.$$

By an argument using the universal type T_{univ} it can be shown that (proof omitted):

Proposition 6.10: Let $i \geq 1$ and $f : \mathbf{N} \to \mathbf{N}$. Then for each query $Q \in \mathrm{CALC}_{0,i}$ there is a query $Q' \in \mathrm{CALC}_{0,1}$ and a $c \in \mathbf{N}$ such that $Q|_f \equiv Q'|_g$, where $g(n) = hyp(c, f(n), i)$. □

We now turn to the expressive power of finite and countable invention. To begin with, let us first consider the classes $\mathrm{CALC}_{0,0}^{fi}$ and $\mathrm{CALC}_{0,0}^{ci}$, which provide different semantics for the relational calculus. Note that countable invention in this context is the unlimited interpretation. It turns out that these are not richer than the limited interpretation.

Theorem 6.11: [HS89a] $\mathrm{CALC}_{0,0}^{ci} \equiv \mathrm{CALC}_{0,0}^{fi} \equiv \mathrm{CALC}_{0,0}^{ni}$. □

Hence, $\mathrm{CALC}_{0,0}^{fi} \sqsubset \mathrm{CALC}_{0,1}^{fi}$ and $\mathrm{CALC}_{0,0}^{ci} \sqsubset \mathrm{CALC}_{0,1}^{ci}$. Recall that the class $\mathcal{E}$ of elementary queries is exactly the union of the hierarchy $\mathrm{CALC}_{0,i}^{ni}$. The proof of the following theorem shows that the classes $\mathrm{CALC}_{0,1}^{fi}$ and $\mathrm{CALC}_{0,1}^{ci}$ are much richer than $\mathcal{E}$.

Theorem 6.12: $\mathcal{E} \equiv \bigcup_{0 \leq i} \mathrm{CALC}_{0,i}^{ni} \sqsubset \mathrm{CALC}_{0,1}^{fi} \sqsubset \mathrm{CALC}_{0,1}^{ci}$.

The proof of the above theorem is accomplished by Lemmas 6.13, 6.15, and 6.18 presented below.

From Examples 2.4 and 3.1, we know that for each query $Q = \{t/T \mid \phi\} \in \mathrm{CALC}_{0,i}$, ϕ can be rewritten so that the range of each quantified variable in ϕ will be restricted to the set of objects constructible from the active domain. Suppose $\delta(x)$ states that $adom(x) \subseteq adom(d, Q)$. Then, each subformula $\forall x\, \psi$ of ϕ is replaced by $\forall x(\delta(x) \to \psi)$; and each subformula $\exists x\, \psi$ of ϕ is replaced by $\exists x(\delta(x) \wedge \psi)$. Now let ϕ' be the formula rewritten from ϕ and $Q' = \{t/T \mid \delta(t) \wedge \phi'\}$. It follows that:

$$Q^{ni} \equiv Q'^{fi} \equiv Q'^{ci}.$$

Suppose now that Q is as above and $\delta(x)$ is a formula stating that $adom(x) \subseteq adom(d, Q) \cup y$ where y is a variable of type U and holds a finite subset of $\mathbf{U}$. Then, for the query $Q'' = \{t/T \mid \exists y/U\ (\delta(t) \wedge \phi')\}$, $Q^{fi} \equiv Q''^{ci}$. From this and Example 6.3 we conclude:

Lemma 6.13: $\mathcal{E} \sqsubseteq \mathrm{CALC}_{0,1}^{fi} \sqsubseteq \mathrm{CALC}_{0,1}^{ci}$. □

In the following, we show that the above "containments" are proper. In other words, there are queries under the semantics of finite and countable invention which can not be simulated by any queries under the semantics of limited interpretation and finite invention respectively. The next example shows that using sets, the halting problem is expressible under the semantics of finite invention.

Example 6.14: Consider a Turing machine M. A computation of M consists of the content of the tape and the position of the tape head changing over time. Recall from Example 3.5, the computation can be encoded using a four dimensional array, where a tuple (t, p, r, s) indicates that at time t the tape position p has symbol r and if $s = q_i$ then the tape head is at position p and the state is q_i, $s =$ '$-$' otherwise. Since invention permits the use of new atomic values, a computation can be represented by an object of type $T_{TM} = \{[U, U, U, U]\}$.

Suppose now that M has input alphabet $\{a\}$. There is a query $Q \in \text{CALC}^{fi}_{0,1}$ from $D =< R : U >$ to $T = U$ with the property that for $d =< R : I >$, $Q^{fi}[d] = I$ if M halts on input $a^{|I|}$, and $Q^{fi}[d] = \emptyset$ otherwise. In particular, $Q = \{t/U \mid R(t) \wedge \exists s/[U, U]\, \exists x/T_{TM}\, (\text{ORD}_U(s) \wedge \text{COMP}_{M,T_{TM}}(s, x))\}$, where ORD_U and $\text{COMP}_{M,T_{TM}}$ are from Example 3.5, ORD_U specifies a total order of index elements; and $\text{COMP}_{M,T_{TM}}$ is essentially similar to the one in Example 3.5 but further states that x contains an encoding of a halting computation of M on input $a^{|I|}$. $\square$

Following directly from Example 6.14, Lemma 6.13, and the definition of $\mathcal{E}$, we have:

Lemma 6.15: $\mathcal{E} \sqsubset \text{CALC}^{fi}_{0,1}$. $\square$

In order to show that countable invention is strictly more expressive than finite invention we establish two lemmas.

Lemma 6.16: For each $Q \in \text{CALC}_{0,1}$ the set $\{(d, o) \mid o \in Q^{fi}[d]\}$ is recursively enumerable.

Proof: By the definition of finite invention, $\{(d, o) \mid o \in Q^{fi}[d]\} = \bigcup_{0 \leq n} \{(d, o) \mid o \in Q|_n[d]\}$. Obviously, given any $n \in \mathbf{N}$, query Q, database d, the set $Q|_n[d]$ of objects is decidable (note that in particular $Q|_0[d] = Q^{ni}[d]$). So there exists a Turing machine M which enumerates the objects in the set $\{(d, o) \mid o \in Q^{fi}[d]\}$. $\square$

The following example is used to establish the second lemma.

Example 6.17: Suppose M is a Turing machine, $D =< R : U >$, and $d =< R : I >$ is a database instance of D as in Example 6.14. Define the database mapping $f_{\overline{M}}$ as:

$$f_{\overline{M}}(d) = \begin{cases} \emptyset & M \text{ halts on input } a^{|I|} \\ \{c\} & \text{otherwise} \end{cases}$$

where c is a constant in $\mathbf{U}$. Consider the query $Q \in \text{CALC}_{0,1}$ where

$$Q = \{t/U \mid t = c \wedge \neg \exists s/\{[U, U]\}\, \exists x/T_{TM}\ (\text{ORD}_U(s) \wedge \text{COMP}_{M,T_{TM}}(s, x))\}.$$

Under countable invention, Q^{ci} expresses the mapping $f_{\overline{M}}$, i.e., $Q^{ci}[d] = f_{\overline{M}}(d)$ for any database d. This is because the variable x ranges over all possible computations of M. On the other hand, $Q^{fi} \not\equiv f_{\overline{M}}$ because for each j, the variable x in $Q|_j[d]$ ranges over computations of M with size up to $j + |adom(d, Q)|$. $\square$

The following completes the proof of Theorem 6.12.

Lemma 6.18: There is a Turing machine M such that for each query $Q^{fi} \in \text{CALC}^{fi}_{0,1}$, $Q^{fi} \not\equiv f_{\overline{M}}$. Thus, $\text{CALC}^{fi}_{0,1} \sqsubset \text{CALC}^{ci}_{0,1}$.

Proof: Let M accept a non-recursive subset $L(M)$ of $\{a\}^*$. Suppose that $Q^{fi} \equiv f_{\overline{M}}$. Then, by the definition of $f_{\overline{M}}$ and Lemma 6.16, $\{(d, c) \mid M \text{ does not halt on input } a^{|I|}\}$ $= \{(d, c) \mid Q^{fi}[d] = \{c\}\}$ is recursively enumerable. This implies that both $L(M)$ and $L(M)^c$ are recursively enumerable, whence $L(M)$ is recursive, a contradiction. □

The final topic of the section is the relationship between calculus queries with invention and *computable queries*.

Definition: The class $\mathcal{C}$ of *computable queries* is the set of queries $Q : D \to T$ such that

(a) T and D are flat;

(b) Q is generic; and

(c) Q is Turing-computable (again assuming a fixed encoding of **U** into strings over $\{0, 1\}$).

The above definition of computable queries carries the spirit of computable queries of [CH80] and deterministic computable database mappings of [AV87, AV88]. In our investigation, queries are mappings from databases to relations whereas in [AV87, AV88] these are mappings from and to databases and in [CH80] the output is a set of tuples, not necessarily having fixed (data independent) widths.

Note that finite and countable invention can express queries which are not computable. [Following the spirit of Example 6.17, the mapping f_M, where $f_M(d) = \{c\}$ if M halts on input $a^{|I|}$ and $\emptyset$ otherwise, is expressible using finite invention. Obviously f_M is not Turing computable.] On the other hand, computable queries are in general partial while all calculus queries studied here are total. To examine the situation in more detail, we allow queries to return undefined, denoted "**?**". It is easy to see that under finite invention the calculus with the undefined value can express all computable queries. Hence, $\text{CALC}^{fi}_{0,1}$ and $\text{CALC}^{ci}_{0,1}$ are essentially more powerful than **QL** of [CH80] and **detTL** of [AV88].

Next we introduce a new semantics for the calculus, named *terminal invention*, and show that it has expressive power equivalent to the computable queries.

Definition: Let $Q \in \text{CALC}_{0,i}$ be a query expression mapping from D and d a database instance of D. The value of Q under the semantics of *terminal invention*, denoted Q^{ti}, is defined as:

$$Q^{ti}[d] = \begin{cases} Q|_n[d] & \text{if } n \text{ is least such that for some } Y \text{ with} \\ & |Y - adom(d, Q)| = n,\ Q|^Y[d] \text{ contains an invented value} \\ \textbf{?} & \text{if there is no such } n \end{cases}$$

The family of such queries is denoted as $\mathrm{CALC}^{ti}_{0,i}$.

To give the intuition behind the role of n in the above definition, we briefly indicate how queries under terminal invention can be used to simulate Turing machines. In particular, a query Q_M can be defined such that (a) if the computation of M doesn't halt within k steps, then $adom(Q|^Y[d]) \subseteq adom(d,Q)$ for each Y with $|Y - adom(d,Q)| \leq k$; and (b) if M does halt with k steps, then $adom(Q|^Y[d]) \not\subseteq adom(d,Q)$. Thus, successful completion of the simulation of M can be monitored by checking the condition $adom(Q|^Y[d]) \subseteq adom(d,Q)$. In this context, the n of the definition is least such that the Turing simulation successfully completes. (Other mechanisms for monitoring successful completion could be devised, and would yield a semantics with equivalent expressive power.)

Note that the $\mathrm{CALC}^{ti}_{0,i}$ hierarchy also collapses at level 1 by reasoning similar to the proof of Lemma 6.5. The following theorem shows the equivalence of terminal invention and the class of computable queries.

Theorem 6.19: $\mathrm{CALC}^{ti}_{0,1} \equiv \mathcal{C}$, i.e., $\mathrm{CALC}^{ti}_{0,1}$ is equivalent to the class of computable queries.

Proof: From the definition, it is obvious that for each $Q \in \mathrm{CALC}_{0,1}$ and database instance d, $Q|_n[d]$ is computable and hence Q^{ti} is computable. Now let f be a generic and computable database mapping. Then, there is a finite $C = \{c_1, ..., c_r\}$ such that f is C-generic and a Turing machine M which computes f. Without loss of generality, we assume that M uses constants in C explicitly and also symbols $\sigma_1, ..., \sigma_p$ which will be used for encoding of other domain elements, i.e., the alphabet $\Sigma = \{c_1, ..., c_r, \sigma_1, ..., \sigma_p\}$.

Now the construction of the calculus query comes down to encoding domain elements into Σ^*; simulating the behavior of M; and decoding the output of M back to elements of $\mathbf{U}$. The encoding of domain elements into Σ^* is analogous to the encoding used in the proof of Corollary 4.6. In particular, there is a formula $\xi_{\mathrm{ENC}}(s,w)$ in the calculus which states that s is a total ordering of all relevant atomic objects and w holds encodings of elements in the active domain of the input.

Similar to Examples 3.5 and 6.14, we use $T_{TM} = \{[U,U,U,U]\}$ to encode a (partial) computation of M. Suppose x of type T_{TM} holds an encoding of a computation of M with an total order s (of indices). Using the formula $\mathrm{COMP}_{M,T_{TM}}$ of Example 3.5, it is straightforward to write a formula $\phi_{\mathrm{SUC}}(s,w,x,z)$ which states that, when s holds a total order and w holds an encoding of the active domain of the input database into Σ^*:

1. x holds a halting computation of M starting with the encoding (according to w) of the input database; and
2. z holds the output of the execution of M, presented as a list of tuples.

Finally, we use the query

$$Q = \{t/T \mid \exists s/\{[U,U]\}\, \exists w/\{[U,U,U,U]\}\, \exists x/T_{TM}\, \exists z/\{[U,U]\}\, \psi\}$$

where

$$\psi = \mathrm{ORD}_U(s) \wedge \xi_{\mathrm{ENC}}(w) \wedge \phi_{\mathrm{SUC}}(s,w,x,z) \wedge (\phi_{\text{IS-IN}}(t,z,w) \vee \neg\delta(t))$$

and where $\phi_{\text{IS-IN}}(t,z,w)$ states that (relative to w) t occurs in the listing z, and $\delta(t)$ states that each atom in t is in the active domain; and $\mathrm{ORD}_U(s)$ states that s holds a total order on $cons_{_{adom(d)}}(U)$. It is then clear that Q^{ti} expresses f. □

As with Turing machines, a notion of "nondeterminism" [AV87] arises naturally in the context of database mappings and a class of "nondeterministically computable queries" was recently proposed in [AV88]. In the following, we show an interesting relationship between the family of non-deterministic computable database mappings and $\mathrm{CALC}^{fi}_{0,1}$. Let $\mathrm{CALC}_{<0,1>,1}$ denote the family of queries in $\mathrm{CALC}_{1,1}$ with intermediate types from τ_1 which map from set-height 0 database schemas to set-height 1 types of the form $\{[U,...,U]\}$. Thus, the output can be viewed as a set of flat relations of the same type. We view the semantics of a query to be any relation in that set, i.e., for $Q \in \mathrm{CALC}^{fi}_{<0,1>,1}$ define an associated non-deterministic mapping f_Q from relational database schemas to relations to have the graph $\{(d,I) \mid I \in Q^{fi}[d]\}$. The family is essentially equivalent in expressive power to **WTL** of [AV88]. Formally, we state without proof the following theorem:

Theorem 6.20: The family $\{f_Q \mid Q \in \mathrm{CALC}^{fi}_{<0,1>,1}\}$ is equivalent to the class of nondeterministic computable database mappings. □

7 Conclusions

In this paper we focus primarily on the expressive power that the complex object query languages bring to the relational model. In particular, we introduce and study the hierarchy of families $\mathrm{CALC}_{0,i}$ of queries which lies between the relational calculus and the computationally complete languages. This study is particularly important in view of the strong interest in the database community in query languages which permit the use of the set construct, both in query inputs and outputs, and more subtly in the intermediate types of queries. Furthermore, it establishes a framework for analyzing query languages for the relational model whose expressive power stands above the second order queries (e.g. SO), and more generally for analyzing complex object query languages. We also studied the $\mathrm{CALC}_{0,i}$ hierarchy under semantics which permit "invention" of temporary objects. This gave insight into various encoding mechanisms for complex objects, and yielded families of queries which were more powerful than the class of computable database queries.

The notion of intermediate types raises a style of question which can be asked in a broad variety of contexts. For example, several proposals for deductive database languages which use complex objects have been made [AG88, BNR$^+$87, Kup87]. The complexity results of Section 4 can easily be extended to yield analogous results for

these languages. It also appears that the analog of the Hierarchy Theorem of Section 5 holds, but no proof is known. A related topic is to study the $\mathrm{CALC}_{0,i}$ hierarchy extended with a least fixpoint operator (see [AB88, GvG88]). For instance, it is known that the family of first order queries is weaker than the family of first order fixpoint queries which in turn is weaker than the family of second order queries. It is likely that the set construct is still more expressive than fixpoint in the context of $\mathrm{CALC}_{0,i}$ for $i > 0$. Another interesting direction is to consider languages weaker than the complex object calculus. For example, recall the *nest* and *unnest* operators from the algebra for nested relations [FT83, JS82]. Let ALG^- denote the algebra for nested relations which includes the usual operators and nest and unnest, but not the powerset operator. It is shown in [PvG88] that the $\mathrm{ALG}^-_{0,i}$ collapses, and that $\cup_{0\leq i}\mathrm{ALG}^-_{0,i} \equiv \mathrm{CALC}_{0,0}$, i.e., the relational calculus. Finally, this approach has also been applied to analyze the expressive power of queries directed towards semantic and object-oriented databases [HS89b].

Appendix: Proof of Proposition 6.9

Proposition 6.9: If objects built using infinite sets are permitted, then for each $i \geq 0$, $\mathrm{CALC}^{ci}_{0,i} \sqsubset \mathrm{CALC}^{ci}_{0,i+1}$.

Proof: The proof uses a diagonal argument. More specifically, we show that for each $i \geq 0$, there is a query $Q \in \mathrm{CALC}_{0,i+1}$ such that no query in $\mathrm{CALC}_{0,i}$ is equivalent to Q under countable invention.

Let i be fixed. The query Q has the input schema $< P : [U, U] >$ consisting of a single binary relation and the output type U. A particular class of input instances are $< P : O_n >$ where O_n is a binary relation such that (1) $|adom(O_n)| = n$, and (2) O_n is a total order on $adom(O_n)$. The query Q on input $< P : I >$ behaves as follows:

1. Q checks if $I = O_n$ for some n. If not return $\emptyset$, otherwise continue;
2. Q builds an encoding of the nth expression E_n (according to some lexicographic ordering) which is potentially the formula of a query in $\mathrm{CALC}_{0,i}$;
3. Q checks if E_n is in fact a well-typed query (in prenex and conjunctive normal form) in $\mathrm{CALC}_{0,i}$ mapping from $< P : [U, U] >$ to U. If not return $\emptyset$, otherwise continue;
4. Q simulates the operation of E_n on input I (under countable invention):
   (1) if $E_n(O_n) = \emptyset$ then return $adom(I)$;
   (2) if $E_n(O_n) \neq \emptyset$ then return $\emptyset$.

It is easily seen that the constructed query Q will not be equivalent to any query in $\mathrm{CALC}_{0,i}$. For the construction of Q, the three main components of Q are listed below:

(a) building and checking the formula E_n;

(b) encoding the set of all assignments. Here the major concern is to find a way to store the set of all possible tuples $[o_1, ..., o_k]$ where k is (roughly) the number of variables in E_n;

(c) checking if any object is in the answer of $E_n(O_n)$.

(a) and (c) can be expressed in the calculus without much difficulty. The remainder of this proof is devoted to the encoding method of (b). The following example illustrates the encoding used.

Example .1: A key aspect of the encoding concerns holding objects of arbitrary width in objects of a fixed width. This will "cost" one level of set-height. To illustrate the spirit of the encoding, let $T = [\{[U, U]\}, U]$, and let $o = [\{[a, b], [a, c]\}, b]$. We encode o into an object of type T_1 in τ_2 whose width is 5 as follows:

$$\left\{ \begin{array}{[ccccc]} i1_1 & c1 & s5 & - & \{[i0_1, c1, a], [i0_1, c2, b], [i0_2, c1, a], [i0_2, c2, c]\} \\ i1_1 & c2 & s4 & b & - \end{array} \right\}$$

The values $i1_1, i0_1, c1$, etc., serve various roles in the encoding, and '$-$' indicates a "don't care" value. The second tuple indicates that column 2 of o holds the value b. Here, the "selector" $s4$ indicates that the fourth column holds the relevant encoded value (selectors si for $i \in [1..3]$ are prohibited). The first tuple indicates the value held by the first coordinate of o. Note that this value is itself stored in encoded form.

In this example, we encoded a 2-tuple of set-height 1 into a set of 5-tuples (with overall set-height 2). The same technique can be used to encode a k-tuple of set-height 1 into T_1 where k is arbitrarily large. Intuitively, this is because the encoding "folds" all of the coordinates of the original object into the fourth and fifth columns of T_1. Furthermore, by using different identifiers in the first coordinate ($i1_2, i1_3$, etc.), an arbitrarily large number of set-height 1 objects can be packed into a single object of type T_1. □

We assume that all underlying types have the tuple construct as root. The following definition formalizes and generalizes the type T_1 used in the above example.

Definition: We define $T^{\text{column}} = U$ and $T^{\text{select}} = U$. The types $T_j^{\text{obj-id}}$ and T_j ($j \geq 0$) are recursively defined as:

1. $T_0^{\text{obj-id}} = U$ and $T_0 = \{[T_0^{\text{obj-id}}, T^{\text{column}}, U]\}$;
2. $T_{j+1}^{\text{obj-id}} = \{T_j^{\text{obj-id}}\}$ and $T_{j+1} = \{[T_{j+1}^{\text{obj-id}}, T^{\text{column}}, T^{\text{select}}, T_0, T_1, ..., T_j]\}$.

Further, let $T_j^{\text{ass-id}} = T_j^{\text{obj-id}}$, $T^{\text{var-id}} = U$, and for each $j \geq 0$,

$$\widehat{T}_j = \{[T_j^{\text{ass-id}}, T^{\text{var-id}}, T_j^{\text{obj-id}}, T^{\text{column}}, T^{\text{select}}, T_0, T_1, ..., T_{j-1}]\}.$$

If E_n encodes a query in $\text{CALC}_{0,i}$, then the type $\widehat{T}_i$ will be used hold the set of all assignments of variables in E_n. Using the assumptions of countable invention and infinite objects, it is easily verified that for each $j \in [0..i]$, there is a sufficient number of objects in $T_j^{\text{obj-id}}$ to accomplish the encodings. □

Acknowledgment

We thank Ron Fagin for suggesting the use of Bennett's thesis in connection with our analysis of the $\mathrm{CALC}_{0,i}$ hierarchy. We also thank Victor Vianu both for inspiring the notion of terminal invention, and for detailed comments and suggestions leading to numerous improvements in our exposition.

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.

[Abr74] J.R. Abrial. Data semantics. In *Data Base Management*, pages 1–59. North-Holland, Amsterdam, 1974.

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, CA, 1988.

[AG88] S. Abiteboul and S. Grumbach. COL: A Logic-Based Language For Complex Objects. In *Advances in Database Technology - EDBT '88*, Lecture Notes in Computer Science. Springer-Verlag, 1988.

[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 (in Russian). *Proc. of USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986. (Translated by Dina O. Goldin, Brown University, October, 1986).

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[AU79] A.V. Aho and J.D. Ullman. Universality of Data Retrieval Languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[AV87] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification. In *Proc. ACM Symp. on Principles of Database Systems*, pages 260–268, 1987.

[AV88] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *Proc. ACM Symp. on Principles of Database Systems*, pages 240–250, 1988.

[Ben62] J.H. Bennett. *On Spectra*. PhD thesis, Princeton University, Princeton, N.J., 1962.

[BK89] F. Bancilhon and S. Khoshafian. A Calculus for Complex Objects. *Journal of Computer and System Sciences*, 38(2):326–340, April 1989.

[BNR+87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Schmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM Symp. on Principles of Database Systems*, 1987.

[CH80] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[CH82] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[CH85] A.K. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 1(1):1–15, 1985.

[Cha81] A.K. Chandra. Programming primitives for database languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 50–62, 1981.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS Proceedings, 1974.

[FT83] P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE Computer Software Applications Conference*, pages 464–475, 1983.

[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 Computing Surveys*, 19(3):201–260, 1987.

[HM81] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Trans. on Database Systems*, 6(3):351–386, 1981.

[HS88] R. Hull and J. Su. On the Expressive Power of Database Queries with Intermediate Types. In *Proc. ACM Symp. on Principles of Database Systems*, pages 39–51, 1988.

[HS89a] R. Hull and J. Su. Domain independence and the relational calculus. Technical Report 88-64, Computer Science Dept, Univ of Southern California, 1989.

[HS89b] R. Hull and J. Su. On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 147–158, 1989.

[HS89c] R. Hull and J. Su. Untyped Sets, Invention, and Computable Queries. Technical Report 89-08, Computer Science Department, Univ of Southern California, 1989. An extended abstract appeared in *Proc. of ACM Symp. on Principles of Database Systems, 1989*.

[Hul86] R. Hull. Relative information capacity of simple relational schemata. *SIAM Journal on Computing*, 15(3):856–886, 1986.

[Hul87] R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256. Academic Press (London), 1987.

[HY84] R. Hull and C.K. Yap. The format model: A theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Jac82] B. Jacobs. On Database Logic. *Journal of the ACM*, 29(2):310–332, April 1982.

[JS82] B. Jaeschke and H.J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. ACM Symp. on Principles of Database Systems*, 1982.

[Kol87] P.G. Kolaitis. The expressive power of stratified logic programs, 1987. manuscript, Stanford University.

[Kup87] G.M. Kuper. Logic programming with sets. In *Proc. ACM Symp. on Principles of Database Systems*, pages 11–20, 1987.

[KV84] G.M. Kuper and M.Y. Vardi. A New Approach to Database Logic. In *Proc. ACM Symp. on Principles of Database Systems*, pages 86–96, 1984.

[KV88] G.M. Kuper and M.Y. Vardi. On the Complexity of Queries in the Logical Data Model. In M. Gyssens, J. Paredaens, and D. van Gucht, editors, *ICDT'88 - Proc. 2nd int. conf. on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pages 267–280. Springer-Verlag, 1988.

[Mai83] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Potomac, Maryland, 1983.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relations in the relational data model. In *Proc. Int. Conf. on Very Large Data Bases*, pages 447–453, 1977.

[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.

[Rei80] R. Reiter. Equality and Domain Closure in First-Order Databases. *Journal of the ACM*, 27(2):235–249, April 1980.

[RKS88] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988.

[Shi81] D.W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, 1981.

[Sto77] L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1977.

[Ull82] J.D. Ullman. *Principles of Database Systems (2nd edition)*. Computer Science Press, Potomac, Maryland, 1982.

[Var82] M.Y. Vardi. The Complexity of Relational Query Languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[vG88] A. van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann Publishers, Los Altos, CA, 1988.