On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions*

(Extended Abstract)

Richard Hull and Jianwen Su

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

Hull@cse.usc.edu JSu@cse.usc.edu

A formal framework for studying the expressive power and complexity of OODB queries is developed. Three approaches to modeling sets are articulated and compared. The class of *regular* OODB schemas supports the explicit representation of set-valued types. Using an *object-based* semantics for sets, the regular schemas correspond to most implemented OODB systems in the literature; a *value-based* semantics for sets is also introduced. Without restrictions, both of these approaches support the specification of all computable queries. Assuming that the new operator is prohibited, the query language of the regular OODB schemas under the object-based semantics is complete in PSPACE; and under the value-based semantics it has hyper-exponential complexity. The third approach to modeling sets is given by the *algebraic OODB* model, in which multi-valued attributes rather than set-valued types are supported. method implementations can use operators stemming from the relational algebra, and do not have side-effects. The query language of algebraic OODBs is more powerful than the relational algebra but has complexity bounded by PTIME. The expressive power and complexity of data access for other variations of OODBs are also considered. Finally, a new relational query language, called *algebra + pointwise recursion*, is introduced. This is equivalent to the algebraic OODB language, and can compute generalized transitive closure.

*This work supported in part by NSF grant IRI-87-19875 and a grant from AT&T. The first author is also supported by DARPA contract MDA903-81-C-0335. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of NSF, AT&T, DARPA, the U.S. Government, or any other person or agency connected with them.

†Part of this work was performed while the authors were visiting the Information Sciences Institute of USC.

1 Introduction

Databases are used for storing data, for accessing data (querying), and for performing transactions on the data. Traditional database models (including the relational and many semantic models) separate the static aspects of databases from the dynamic aspects, primarily by defining an essentially static database *schema*, and separately defining query and transaction languages. On the other hand, a central aspect of the object-oriented paradigm in the context of databases [BCG+87, PSM87, LRV88, AH87b] is that it provides a natural framework whereby dynamic aspects of a database can be incorporated directly into a database schema (in the form of methods). In this paper we initiate a theoretical investigation into the impact of this approach on data access by studying the expressive power of queries directed towards object-oriented databases (OODBs).

The paper makes four fundamental contributions. First, we introduce a formal framework for studying the expressive power and complexity of OODB queries. A query in this model consists in both a family of methods for the underlying types, and a procedural "external" query; this follows the approach adopted by [AH87b, BCD89], and to a lesser extent, that of [PSM87]. The expressive power of OODB queries is compared with that of queries against relational databases by using a natural correspondence between OODB schemas and relational schemas; complexity is also defined in a straightforward manner.

The second contribution of the paper consists in the articulation and comparison of three alternative ways to incorporate sets into OODBs. In particular, we consider two ways in which set-valued types are explicitly represented in the schema; the first of these uses an *object-based* semantics for sets as in the ORION [BCG+87] system, and the other uses a *value-based* semantics which is

motivated from the perspective of "complex database objects" (essentially, nested relations) [Hul87]. The third approach, called *algebraic* OODBs, prohibits explicit set-valued types, and uses multi-valued attributes (and methods) in their place. Under this approach, method implementations use operations taken from the relational algebra to provide a natural syntax for set manipulation. Also, methods used by queries do not have side-effects.

The third contribution of the paper is a body of theoretical results concerning the expressive power and complexity of data access to OODBs which incorporate sets in these three ways. Not surprisingly, if no restrictions are imposed than all computable queries can be expressed in OODBs under the first two approaches. If the **new** operator is prohibited in the methods used by queries, then a hierarchy of complexities is obtained: (speaking informally) (1) with explicit set-valued types and the value-based set semantics, queries with arbitrarily high hyper-exponential complexity can be expressed; (2) with explicit set-valued types and the object-based set semantics, the access language is complete in PSPACE; and (3) queries on algebraic OODBs are subsumed by PTIME (and are thus tractible).

The final contribution of the paper is to introduce a new relational query language, called *algebra + pointwise recursion*. This is equivalent to the algebraic OODB language, and is closely related to the *traversal recursion* language of the PROBE project [RHDM86]. In particular, it can perform the "generalized transitive closure" operation. It is shown here to be subsumed by the relational calculus with fixpoint in terms of expressive power. (It remains open whether the inclusion is proper).

The focus of this paper is on the expressive power and complexity of data access languages on OODBs. For this reason we make many simplifying assumptions in the formal model (e.g., in the areas of update operators and run-time errors) that we use for the study. Importantly, our model is subsumed by most implemented OODBs, and the simplifications do not materially affect the spirit of our results.

A large portion of this paper is devoted to explicit definitions of various aspects of OODB, including the notions of *schema* and *method*. The definitions are presented here because the field of OODBs is so young that there is no readily available, widely accepted formalism that we could have used for this theoretical investigation. Portions of the basic model were greatly influenced by [HTY88]. Due to space limitations the presentation here is terse and in some cases presents only sketches of the actual definitions. It is assumed that the reader is familiar with the basic concepts of OODBs, the relational model, and the relational algebra and calculus.

Section 2 reviews relational query languages. In section 3, we formally introduce the general model of OODBs used here, and define *regular* OODB schemas, which support explicit set-valued types. An external access language for OODBs is introduced in Section 4, along with a framework of comparing OODB queries and relational queries. Section 5 presents results on regular OODB access languages, under both the object-based and valued-based semantics for sets. We define and study algebraic and other restricted OODB schemas in section 6. The algebra + pointwise recursion is introduced in section 7.

2 Preliminaries

This section present some preliminary concepts. All languages considered are capable of returning the *undefined* value, denoted by '?'.

Definition: Let S be a relational schema and R a relation name. A *relational mapping* from S to R is a total function f from the set of instances of S to the set of instances of $R \cup \{?\}$ which is *generic* i.e.[1], $f \circ \sigma = \sigma \circ f$ for each permutation σ over **U**.

Genericity corresponds to the intuition that the objects occurring in databases are essentially "uninterpreted". The notion of genericity can be generalized to incorporate a total order on **U**, by considering only permutations σ which are order-preserving.

Let f be a relational mapping. The time (space) complexity of f is the time (space) used to decide $o \in f(d)$ for any relational instance d and tuple o. A set F of relational mappings (a language L) is said to be *C-complete* for some complexity class C if every set in C is polynomial reducible to some $f \in F$.

We now describe several classes of queries. The family of *hyper-exponential* functions hyp_k on **N** are defined recursively as follows: $hyp_0(i) = i$, and $hyp_{k+1}(i) = 2^{hyp_k(i)}$ for each $k \geq 0$. The family of *elementary* queries [HS88a], denoted $\mathcal{E}$, is the set of relational mappings which have time (or space) complexity hyp_k for some $k \in \mathbf{N}$. The family $\mathcal{C}$ of *computable* queries [CH80] contains relational mappings computable by Turing machines. Obviously, $\mathcal{E} \subset \mathcal{C}$.

Let CALC+**fixpt** denote the set of queries expressed in the relational calculus with a least fixpoint operator [CH82] and PTIME (PSPACE) denote the class of deterministic polynomial time (space) recognizable sets. It is known that CALC+**fixpt** is in PTIME [CH82], and that if there is a total order on the underlying domain then the family of mappings expressed by CALC+**fixpt** is precisely polynomial time relational mappings[Imm86, Var82].

Finally, we consider the relational algebra extended with a **while** looping construct, which is denoted as ALG+**while**. It is known that with or without order, CALC+**fixpt** is less expressive than ALG+**while**[CH82]; and it is open whether this inclusion is proper. If there is an underlying order, then ALG+**while** is precisely the set of all database mappings in PSPACE[Var82]. With or without order, $\mathcal{E}$ properly contains ALG+**while**[HS88a].

[1] σ is extended naturally to databases.

3 An Abstract Model for OODBs

This section presents a formal framework for OODBs. Because of our focus on data access, it is natural to view the structural portion (i.e., the type hierarchy) of the OODB as essentially static. For this reason, we begin with definitions of *semantic* database schemas and instances (Subsection 3.1), and then incorporate methods to form *object-oriented* database schemas (Subsection 3.2). We then define the family of *regular* OODB schemas, which capture the spirit of several implemented OODB systems in the literature (Subsection 3.3). Subsection 3.4 briefly indicates the formal definition for the operational semantics of method calls in OODB schemas.

3.1 Semantic schemas and instances

In this subsection we introduce the notions of *semantic* database schemas and instances. Our definition of semantic schemas will follow the general spirit of IFO [AH87a] and FDM [Shi81]. Notably, in this section and the following two sections we do not permit multi-valued attributes; this is not a limitation on the expressive power of the model because set-valued types are permitted. Multi-valued attributes are studied in Section 6.

In our current context, a semantic schema consists of types, attribute (name)s, and ISA relationships. ISA relationships on set-valued types will be restricted in a natural fashion. In the formal model, we assume the existence of 2 disjoint countably infinite sets[2]

- $\mathcal{T}$ of *(abstract) type (name)s*,
- $\mathcal{A}$ of *attribute (name)s*.

In a given semantic schema, some types will be *atomic*, while others will be *set-valued*. The inclusion of set objects introduces a number of possiblities for assigning semantics to the set operations in methods, and indeed, defining the set of valid instances. For the model introduced in this section we have chosen an *object-based* semantics for instances: two distinct objects may have associated sets which are equal. This is closely related to the semantics of most OODB systems in the literature. In Section 5 we introduce an alternative, *value-based* semantics for sets in regular OODB schemas.

Definition: A *semantic (database) schema* is an ordered 4-tuple $S = (T, STR, ISA, Att)$ where

1. T is a finite subset of $\mathcal{T}$.
2. STR, the *set type restriction* function, is a partial function from T to T whose graph is acyclic. The domain of STR is denoted T_{set}; and T_{atom} denotes $T - T_{\text{set}}$. Elements of T_{set} are *set-valued* types; elements of T_{atom} are *atomic* types.

[2] In practical systems it is typical to include a family of *basic* types, including, e.g., integer, boolean, etc. Although their incorporation would change some aspects of our model, it would not materially affect the spirit of our results.

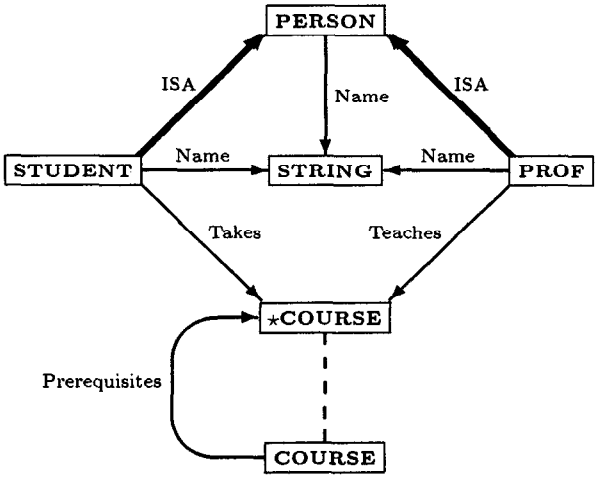

Figure 1: An Example of Semantic Schema

3. (T, ISA) is a directed forest (with edges pointing from child to parent) such that:
   (a) if $(s,t) \in ISA$ then either both s and t are in T_{atom}, or both s and t are in T_{set}.

   s is a *subtype* of t (t is a *supertype* of s), denoted $s \preceq t$ ($t \succeq s$), if there is a path from s to t in (T, ISA). Also,
   (b) if $s, t \in T_{\text{set}}$ with $STR(s) = s'$ and $STR(t) = t'$, then $(s,t) \in ISA$ implies $s' \preceq t'$.
4. $Att : T \rightarrow \mathcal{P}^{fin}(\mathcal{A} \times T)$. Intuitively, Att assigns attributes and attribute ranges to types, i.e., if $(a, t') \in Att(t)$, then attribute a is defined on type t and has range t'. Att satisfies the following restrictions:
   (a) $(s,t) \in ISA$ and $(a,t') \in Att(t)$ implies $(a,s') \in Att(s)$ for some $s' \preceq t'$, and
   (b) $(a,t'), (a,t'') \in Att(t)$ implies $t' = t''$

This formalism incorporates a number of assumptions which are motivated more completely in the full paper. Notably, multiple inheritance is not permitted, and nested sets are supported.

Example 3.1: Figure 1 presents a semantic schema in graphic form. The dashed line between "⋆COURSE" and "COURSE" denotes the set type restriction; bold faced vectors are ISA edges; and the remaining vectors are attributes. □

We next define the notion of database instance of a semantic schema. For this definition we assume the existence of a countably infinite set

- $\mathcal{O}$ of *abstract objects*.

Definition: Let $S = (T, STR, ISA, Att)$ be a semantic schema. A *(semantic database) instance* of S is a triple $d = (O, SV, A)$ where

- $O : T \rightarrow \mathcal{P}^{fin}(\mathcal{O})$, such that if $s \neq t$ then $O(s) \cap O(t) = \emptyset$.

Using O, we define the function $O^* : T \rightarrow \mathcal{P}^{fin}(\mathcal{O})$ so that $O^*(t) = \cup\{O(s) \mid s \preceq t\}$.

- SV is a total function with domain $\cup\{O(t) \mid t \in T_{\text{set}}\}$, such that for each $t \in T_{\text{set}}$, $SV|_{O(t)}$ is a 1-1 total function mapping each object in $O(t)$ to a subset of $O^*(STR(t))$

(Intuitively, SV is the *set-value* function, which assigns a set of appropriate type to each object in a set-valued type.)

- A is a *partial* function[3] with domain $\{(o,a) \mid \exists t, t' \in T$, such that $o \in O(t)$ and $(a,t') \in Att(t)\}$. For each such pair (o,a) with associated types t, t', $A(o,a) \in O^*(t')$. We generally use $o.a$ to denote $A(o,a)$.

For $t \in T$ we often write $d(t)$ $(d^*(t))$ to mean $O(t)$ $(O^*(t))$.

Example 3.2: Consider the schema in Example 3.1. The following O, SV, and A yield and a semantic instance:

$O(\text{PERSON}) = \{p_1\}$	$O(\text{STRING}) = \{n_1, ..., n_4\}$
$O(\text{STUDENT}) = \{p_2, p_3\}$	$O(\text{COURSE}) = \{c_1, c_2, c_3\}$
$O(\text{PROF}) = \{p_4\}$	$O(\star\text{COURSE}) = \{s_1, s_2\}$
$SV(s_1) = \{c_1, c_2\}$	$SV(s_2) = \{c_1, c_3\}$
$A(p_j, \text{Name}) = n_j$	$A(p_4, \text{Teaches}) = s_2$
$A(p_i, \text{Takes}) = s_{i-1}$	$A(c_i, \text{Prerequisites}) = c_{i-1}$

where $j \in [1..4]$ and $i \in [2..3]$. □

3.2 OODB schemas and instances

In our formalism, an object-oriented database schema is essentially a semantic schema with methods. Notably, methods here will not be able to introduce new *types* into a schema.

We assume the existence of 2 additional disjoint countably infinite sets

- $\mathcal{M}$ of *method (name)s*,
- $\mathcal{I}$ of *implementations*.

Implementations are fragments of executable code; these will be discussed in more detail below. Methods are abstract names which will be used (in the context of a given schema and given type) to refer to implementations.

Definition: An *object-oriented (database) schema* (OODB schema) is an ordered 6-tuple $\widehat{S} = (T, STR, ISA, Att, Meth, Impl)$ where

1. $S = (T, STR, ISA, Att)$ is a semantic schema.
2. $Meth : T \rightarrow \mathcal{P}^{fin}(\mathcal{M})$, such that: if $(s,t) \in ISA$ and $m \in Meth(t)$, then $m \in Meth(s)$.
3. $Impl$ is a partial function from $\{ (m,t) \mid m \in Meth(t) \} \rightarrow \mathcal{I}$, such that: if $m \in Meth(s)$ then there is some $t \succeq s$ such that $Impl(m,t)$ is defined.[4] The function $Impl$ is also subject to other restrictions, which are described later.

In this case, $\widehat{S}$ is called an *OODB extension* of S, and S is called the *semantic schema* of $\widehat{S}$. We also define an induced function $Impl^*$:$\{ (m,t) \mid m \in Meth(t) \} \rightarrow \mathcal{I}$. In particular, $Impl^*(m,s) = Impl(m,t)$, where t is the least supertype of s for which $Impl(m,t)$ is defined. (The restriction of (3) above guarantees the existence of such a t.) Note that $Impl^*$ is a total function on $\{ (m,t) \mid m \in Meth(t) \}$. An *(OODB) instance* of $\widehat{S}$ is an instance of the semantic schema S.

3.3 Regular implementations

In general, an object-oriented database model provides an imperative language for specifying implementations for methods. (This contrasts with "pure" object-oriented programming languages such as SMALLTALK, where even the most primitive operations are accomplished by method calls.) The imperative language may be an existing language (e.g., C), or might be developed specifically for the model. In any case, the language typically has special primitives for working with objects, their attributes, and in the case of set-types, the memberships of objects. In the current discussion we study families of schemas in which the implementations use specific, restricted languages. In this section we present one of these languages, which will be appropriate for studying the expressive power and complexity of data access to most currently implemented OODB systems.

In the following, the set **par** of variables refers to objects passed as parameters in the method call, and the set **var** holds internal variables which range over objects.

Definition: An implementation is said to be *regular* if it has the following form:

$$\textbf{par: } u_1, \ldots, u_n; \quad \textbf{var: } w, x, y, \ldots, z;$$
$$s_1; \ldots; s_k; \quad \textbf{return}(x)$$

where

1. $n \geq 0$, $m \geq 1$, $l \in [1..m]$; and
2. s_p has any of the following forms for $p \in [1..k]$:

basic operations:

(a) $w :=$ self

(b) $w :=$ self.a; where $a \in \mathcal{A}$

(c) $w := x$

(d) $w := m(x; y, \ldots, z)$ (where $m \in \mathcal{M}$)

(e) self.$a := w$; where $a \in \mathcal{A}$

type operations:

[3] The use of partial rather than total functions simplifies the issue of inserting new objects into instances.

[4] It is possible to have types s and t and a method m such that $m \in Meth(s)$ and $m \in Meth(t)$, but there is no common supertype r of s and t such that $m \in Meth(r)$.

(f) $w :=$ **new**(t); where $t \in T$ (for $t \in T_{\text{set}}$, $SV(w)$ is initialized to $\emptyset$)

(g) **delete**(t, w); where $t \in T$

set operations: (applicable only to objects in set-valued types)

(h) **set_insert**(w) (which replaces SV(self) by SV(self) $\cup\{w\}$)

(i) **set_delete**(w) (which replaces SV(self) by SV(self) $-\{w\}$)

(j) **set_equal**(z) (which replaces SV(self) by $SV(z)$)

looping operations:

(k) **for each** w **in** t **do** $s'_1; \ldots; s'_l$ **end**; where t is a type and $s'_1; \ldots; s'_l$ is a sequence of statements from this list

(l) **for each** w **in** x **do** $s'_1; \ldots; s'_l$ **end**; where $s'_1; \ldots; s'_l$ is a sequence of statements from this list in which x does not occur (a runtime error occurs if x is not a set object)

conditionals:

(m) **if** $x_i\ \theta\ x_j$ **then** s; where θ is[5] $=_o$, $\neq_o$, $=_v$, $\neq_v$, $\in$, or $\notin$; and s is any nonconditional statement in this list

(n) **if** $x_i\ \theta\ t$ **then** s; where θ is $\in$ or $\notin$, and t is a type; and s is any nonconditional statement in this list

A *regular (OODB) schema* is a schema $\widehat{S} = (T, STR, ISA, Att, Meth, Impl)$ such that $Impl(t, m)$ is a regular implementation for each (t, m) on which $Impl$ is defined; and $\widehat{S}$ satisfies a number of natural **well-formed requirements** (Intuitively, these focus on conditions which are "easily" checked at compile time, with reference only to structural and syntactic properties of types and implementations, and are detailed in the full paper). The family of regular OODB schemas is denoted by $\mathcal{F}_{\text{reg}}$.

We make the following observations about regular schemas: **(1) delete** will be given the usual semantics; note that applying delete may lead to a dangling reference, and thus (as detailed in the next subsection) to an aborted method execution. **(2)** The execution of looping statements is non-deterministic, in the sense that the particular order in which s is performed on members of the looping set is not guaranteed by the implementation. It is the programmer's responsibility to ensure that the execution of the system is deterministic, if deterministic operation is desired. Alternatively, the order of execution could be based on a default ordering of the underlying objects (surrogates) in $\mathcal{O}$. **(3)** We do not provide a primitive for referring to an implementation in a supertype (such as 'super' in SMALLTALK or '$$' in VBASE [AH87b]). **(4)** We have provided a minimal set of operators; others (e.g., while loops) can easily be simulated with these.

[5]Both *object-based* and *value-based* equality is provided; the value-based version is applicable only to set-valued objects.

Example 3.3: We extend the semantic schema in Example 3.1 to a regular OODB schema by associating three methods "`GetName`", "`AllPrereq`", and "`Union`" with PERSON, COURSE, and ⋆COURSE (respectively), with the following implementations:

```
GetName:   var: x;
           x := self.Name;
           return(x)

AllPrereq: var w, x, y, z;
           x := self.Prerequisites;
           w := new(⋆COURSE);
           w := Union(w, x);
           for each y in x do
             z := AllPrereq(y);
             w := Union(w, z);
           end;
           return(w)

Union:     par: u; var x;
           for each x in u do
             set_insert(x);
           end;
           return(self)
```

□

3.4 Method execution trees

In this subsection we briefly indicate the operational semantics provided for method calls on OODB instances. Due to space limitations, the presentation here is extremely brief; the complete definition is provided in the full paper.

This semantics is based on the "Method Execution Trees (METs)" of [HTY88], and provides a faithful copy of the semantics given to method calls in actual object-oriented systems. The semantics is defined in terms of a labeled tree (where nodes have ordered children); the usual operational semantics is obtained from a MET by performing a depth-first traversal of it. The labels hold "method call summaries", which are triples of one of the following forms:

completed:	$\langle (d_{in}, o_{in}, \vec{p}), (m, t), (d_{out}, o_{out}) \rangle$;
aborted:	$\langle (d_{in}, o_{in}, \vec{p}), (m, t), (-, -) \rangle$;
uncalled:	$\langle (-, -), (m, -), (-, -) \rangle$.

Intuitively, m is the method to be called, on the instance d_{in} and object o_{in} with parameters $\vec{p}$. t indicates the type of o; and d_{out} is the resulting instance and o_{out} the returned object (if defined). (We include t as an explicit part of the mcs, because it facilitates the formation of various abstracts of METs.)

There are three kinds of *Method Execution Trees* (METs): *Successful* METs correspond to successful executions of a method call on given input instance, input object, and input parameters. *Aborted* Mets correspond to executions of a method call which are aborted for some reason (e.g., calling a method m' on an object for which

it is not defined, or calling it with the wrong number of parameters; or making a runtime type error, e.g., by attempting to add an element of one type to a set of objects of another type). *Non-terminating* METs correspond to executions of a method call which lead to an infinite sequence of method calls.

4 A Model of Data Access

This section uses a simple correspondence between semantic schemas (and thus, OODB schemas) and relational schemas (Subsection 4.1) to introduce the notion of *OODB access languages* (Subsection 4.2). These languages consist in both a family of OODB schemas and an "external" language for accessing OODB instances. They can be viewed as a formal abstraction of practical access languages such as that provided for O_2 [BCD89], VBASE, and to some extent, GEMSTONE [PSM87]. We conclude the section by introducing formalism for comparing the expressive power and complexity of different access languages.

4.1 A Relational Simulation

We describe here a reduction of semantic schemas to relational schemas; it also applies to the OODB extensions of semantic schemas. If $S = (T, STR, ISA, Att)$ is a semantic schema we associate a relational database schema S_{rel} with the following relations:

1. for each type $t \in T$ a unary relation R_t;
2. for each type $t \in T_{\text{set}}$ a binary relation $R_{t\text{-val}}$;
3. for each pair (t, a) where $(a, t') \in Att(t)$ for some t' a binary relation $R_{t.a}$

There is a natural mapping $flat$ (ten) from instances d of S to instances[6] $flat(d)$ of S_{rel}:

for $t \in T$, $flat(d)[R_t] = \{[o] \mid o \in d(t)\}$
for $t \in T_{\text{set}}$, $flat(d)[R_{t\text{-val}}] = \{[o, o'] \mid o' \in SV(o)\}$
for $(a, t') \in Att(t)$,
$\quad flat(d)[R_{t.a}] = \{[o, o'] \mid o' = o.a \text{ in } d\}$

In this context, we use the set $\mathcal{O}$ of abstract objects as the underlying universal domain for the relations. Thus, the mapping $flat$ makes object IDs directly visible to database users. While this violates the spirit of some general assumptions about using OODBs, it permits us to easily state our results on expressive power, without substantially affecting them. Note also that (in the current context) each instance $flat(d)[R_{t.a}]$ satisfies a functional dependency from the first coordinate to the second coordinate.

Definition: A relational schema S is a *semantic* (or *OODB*) *simulation* if $S = S'_{\text{rel}}$ for some semantic (OODB) schema S'.

[6] In our formalism, a relational instance is a function which maps relation names into sets of tuples.

Example 4.1: The semantic simulation of the semantic instance shown in Example 3.2 has six unary relations, one for each type; one binary relation for the set type ⋆COURSE; and another six binary relations for attributes. The following shows part of this relational database:

R_{PERSON}

p_1
p_2
p_3
p_4

$R_{\star\text{COURSE-val}}$

s_1	c_1
s_1	c_2
s_2	c_1
s_2	c_3

$R_{\text{STUDENT.takes}}$

p_2	s_1
p_3	s_2

□

4.2 Object-oriented access languages

We perform data access on OODBs using an external language, called an "OODB query language", which has the ability to invoke methods inside the OODB schema. Formally,

Definition: An *OODB access language* is a pair $(\mathcal{F}, L)$ where $\mathcal{F}$ is a family of OODB schemas and L is an *OODB query language* for $\mathcal{F}$.

We focus here on an OODB query language which is a slight variant of the relational algebra, called $L_{\text{alg}+\alpha}$. Let $\widehat{S} = (T, STR, ISA, Att, Meth, Impl)$ be a regular OODB schema. A *query* in $L_{\text{alg}+\alpha}$ is a sequence $s_1, \ldots, s_n$,ANS $:= Z$ of assignments having any of the following forms, where the variables range over relations of fixed types, and Z is defined in one of the s_j's (where the appropriate typing constraints are satisfied):

1. $X := R$ for any $R \in \widehat{S}_{\text{rel}}$
2. $X := \psi(Y)$ for any unary relational algebra operator
3. $X := Y_1 \ \psi \ Y_2$ for any binary relational algebra operator
4. **pointwise method application:** $X := \alpha(m, Y)$ where m is a method name (satisfying the type restriction that $arity(Y)$ = the number of parameters of m in $\widehat{S} + 1$, and $arity(X) = arity(Y) + 1$)

The output of an $L_{\text{alg}+\alpha}$ query is a relational instance with arity $arity(Z)$.

Intuitively, if Y holds an n-ary relation and m is a method name with $n - 1$ parameters, then $\alpha(m, Y)$ has the effect of calling m on each tuple y of Y, and returning the relation

$$\{[y_1, \ldots, y_n, x] \mid x \text{ is returned by } m[y_1; y_2, \ldots, y_n]\},$$

if the execution of m on each $\vec{y}$ is successful; and returns ? otherwise. Importantly, this may have side-effects on the underlying OODB instance. Technically, the result of this operation may be non-deterministic, because as with the looping commands it depends on the order of the execution of m on the tuples in Y.

Definition: Let $(\mathcal{F}, L)$ be an OODB access language, and S a semantic schema. A *query* in $(\mathcal{F}, L)$ on S is a pair $(\widehat{S}, Q)$ where $\widehat{S}$ is an OODB extension of S which is in $\mathcal{F}$, and Q is a query in L directed towards $\widehat{S}$.

Recall that relational mappings are required to be *generic*. This implies that for each query f and for each input[7] d, $adom(f(d)) \subseteq adom(d)$. However, the final value of the variable Z in an OODB query in $L_{alg+\alpha}$ may include some newly created objects. The semantics of $L_{alg+\alpha}$ is restricted to prevent this:

Definition: Let S be a semantic schema, and $(\widehat{S}, Q)$ be an OODB query on S where $Q = s_1, \ldots, s_n$,ANS $:= Z$ is in $L_{alg+\alpha}$. For each instance d of S, the output of $(\widehat{S}, Q)$ on d, denoted $(\widehat{S}, Q)[d]$, is

1. ?; if any method invoked during the processing of Q on d with $\widehat{S}$ is not successful, and
2. ANS $\cap$ $adom(d)^{arity(Z)}$, i.e., the value assigned to ANS at the completion of the execution of Q, intersected with the active domain of the input instance; otherwise.

4.3 A Framework for Comparison

"Equivalence" of OODB and relational queries is defined in the natural manner:

Definition: Let S be a semantic schema, $(\widehat{S}, Q)$ an OODB query on S, and Q' a relational query on S_{rel}. Then $(\widehat{S}, Q)$ and Q' are *equivalent*, denoted $(\widehat{S}, Q) \equiv Q'$ and $Q' \equiv (\widehat{S}, Q)$, if for each instance d of S, $(\widehat{S}, Q)[d] = Q'[flat(d)]$.

We now indicate the formal definitions for comparing object-oriented access languages and relational languages:

Definition: Let $(\mathcal{F}, L)$ be an OODB access language and L' a relational query language.

- $(\mathcal{F}, L)$ *subsumes* L', denoted $(\mathcal{F}, L) \sqsubseteq L'$, if for each semantic schema S and each query $(\widehat{S}, Q)$ in $(\mathcal{F}, L)$ for S, there is a query Q' in L' such that $Q' \equiv (\widehat{S}, Q)$.

Subsumption is extended in the natural manner to other combinations of OODB access languages and relational query languages. *Proper subsumption* ($\sqsubset$) and *equivalence* ($\equiv$) are also defined. Note that all comparisons are done relative to relational schemas and instances which are semantic simulations.

If $(\widehat{S}, Q)$ is an OODB query on S, the *complexity* of $(\widehat{S}, Q)$ is the complexity of evaluating $Q(d)$ in terms of the size of an input semantic instance d. Since our focus is on relational mappings, the size of d is always linear to its semantic simulation.

5 Access in Regular OODBs

In this section we examine the expressive power and complexity of OODB access languages based on regular OODB schemas (and using the query language $L_{alg+\alpha}$). We obtain three results, based on variations of two parameters: (a) allowing or prohibiting the **new** operator, and (b) using the "objcet-based" semantics for sets introduced in Section 3 versus using a "value-based" semantics for sets presented below. Theorem 5.1 says that in the case where the **new** operator is used, all computable queries can be expressed. Theorem 5.2 shows that the access language without the **new** operator but using object-based set semantics is PSPACE-complete. On the other hand, when substituting object-based semantics by value-based semantics the access language without **new** has a close relationship with the elementary queries (Theorem 5.4). In particular, queries can be formulated in this context whose space complexity is an arbitrarily large hyper-exponential function. We also comment on data access with other variations of regular OODBs.

In this section all OODB queries are assumed to be from the language $L_{alg+\alpha}$ unless otherwise stated. Recall also that the OODB schemas studied here do not provide primitives for introducing new types into schemas.

The following result essentially shows that $(\mathcal{F}_{reg}, L_{alg+\alpha}) \equiv \mathcal{C}$:

Theorem 5.1:

1. Let S be a semantic schema. Then for each OODB query $(\widehat{S}, Q)$ on S in $(\mathcal{F}_{reg}, L_{alg+\alpha})$, there is a relational query $Q' \in \mathcal{C}$ such that $(\widehat{S}, Q) \equiv Q'$.
2. Let S be a semantic schema with at least one atomic type which has at least four attributes. Then for each computable query Q' (not involving constants) on S_{rel} there is an OODB query $(\widehat{S}, Q)$ on S in $(\mathcal{F}_{reg}, L_{alg+\alpha})$ such that $(\widehat{S}, Q) \equiv Q'$.

Proof: (Sketch) Part (1) is straightforward; and part (2) uses the language detTL of [AV88], in particular showing how the **new** operation in OODBs can be used to simulate the "invented values" of detTL. □

This result applies to most OODB implementations described in the literature. It should be noted that using declarative external languages in place of $L_{alg+\alpha}$ yield more expressive power (see Remark 5.5 at the end of this section).

In view of the above results, it is interesting to consider the expressive power of OODB queries which do not invoke the **new** operator during their execution. To study this, we focus on the following class of OODB schemas:

Notation: $\mathcal{F}_{reg}^{no\text{-}new}$ denotes the family of regular OODB schemas in which the **new** operation does not occur in any implementation.

Theorem 5.2: $(\mathcal{F}_{reg}^{no\text{-}new}, L_{alg+\alpha})$ is PSPACE-complete.

Proof: (Sketch) The proof is based on a reduction from the problem of Quantified Boolean Formulas [GJ79]. □

It remains open how the expressive power of $(\mathcal{F}_{reg}^{no\text{-}new}, L_{alg+\alpha})$ compares with that of ALG+**while**, which is also known to be PSPACE-complete [Cha81].

[7] For an instance d, the *active domain* of d, denoted $adom(d)$, is the set of atomic objects occurring in d.

The above results assumed an "object-based" semantics for sets, in the sense that in order to manipulate a set of values, a (possibly new) set object must be used. In the next result we explore the implications of using a "value-based" semantics for sets. Under this semantics, which is defined more formally in the full paper, the identity of a set is determined exclusively by its membership. It is thus impossible for a set-valued type to hold two distinct objects which correspond to the same set. A set object will be "created" if it is not already present in the database instance, and it is assigned as an attribute value or is the return-value of a method. We assume that set objects are "deleted" from the instance if they are not an attribute value, nor the member of another existing set. Within method implementations, variables can refer to sets which may not be present in the database instance. We modify the set operations of regular implementations to reflect this change in the underlying semantics as follows:

set operations (value-based):

(h′) $x := y \cup \{z\}$;

(i′) $x := y - \{z\}$;

(j′) $x := \emptyset$ **of** t.

In this context, these statements can occur in methods of any (set-valued or atomic-valued) type.

Example 5.3: Let us consider the method `AllPrereq` in Example 3.3. The same output is obtained under the value-based set semantics by the following:

```
AllPrereq′:  var: v, w, x, y, z;
             x := self.Prerequisites;
             w := x;
             for each y in x do
               z := Allprereq′(y);
               for each v in z do
                 w := w ∪ {v};
               end;
             end;
             return(w)
```

□

Notation: $\mathcal{F}_{\text{reg,val}}^{\text{no-new}}$ denotes the family of regular OODB schemas using value-based semantics for set objects in which the **new** operation does not occur in any implementation.

Our main result on $\mathcal{F}_{\text{reg,val}}^{\text{no-new}}$ shows a close relationship between OODB queries based on this class and the family of elementary queries. Our Theorem partitions the family of $\mathcal{F}_{\text{reg,val}}^{\text{no-new}}$ schemas based on their "set-height", and provides a narrow "window" on the maximal complexity for each member of this partition.

Definition: Let $S = (T, STR, ISA, Att)$ be a semantic schema. The *set-height* of S, denoted $sh(S)$, is the largest integer k such that there is a chain of types $t_0, t_1, ..., t_k$ with $t_i \in T$ for $i \in [0..k]$ and $STR(t_i) = t_{i-1}$ for $i \in [1..k]$.

Theorem 5.4: Let $k \geq 1$.

1. Let S be a semantic schema with set-height k. Then the complexity of each OODB query in $(\mathcal{F}_{\text{reg,val}}^{\text{no-new}}, L_{alg+\alpha})$ is at most nondeterministic hyp_{k+1}-time (and hence, at most hyp_{k+1}-space.)
2. There is a semantic schema S with set-height k and an OODB query $(\widehat{S}, Q)$ whose complexity is at least hyp_k-space.

Proof: (Sketch) For part (1), we show that each such query can be simulated by a query in the language $ALG_{0,k}$+**while** ($ALG_{0,k}$ is defined in [HS88b]). In turn, this can be simulated by a query in $CALC_{0,\exists k+2}$ (again see [HS88b]), which by results of [KV88] has complexity $\leq$ nondeterministic hyp_{k+1}-time. For part (2), we build a schema S with a type t whose set-height is k, and which has four attributes. It is shown that this schema can simulate the behavior of all Turing machines which run in hyp_k-space on inputs in which all set-value types are initially empty. □

To summarize intuitively,

space complexity hyp_k
$\leq$ "hardest" OODB query in $(\mathcal{F}_{\text{reg,val}}^{\text{no-new}}, L_{alg+\alpha})$ on semantic schemas with set-height k
$\leq$ space complexity hyp_{k+1}.

Intuitively, the result holds because under the value-based semantics, set formation (either by introducing $\emptyset$ or by adding or deleting elements from existing sets) can implicitly introduce new objects into an OODB instance. The set-height of the underlying semantic schema and the size of the active domain of the input determine the maximum number of new objects that can be created, and thus the amount of "working space" available for simulating Turing machine computations.

The results above focus on OODBs which use a procedural external language and enforce strong typing on set objects. We conclude this section by the following remark which indicates the implications of (1) using a declarative external language in place of $L_{alg+\alpha}$; or (2) relaxing the typing constraints.

Remark 5.5: (1) The OPAL language [PSM87, Ull88] for the Gemstone system includes a declarative language in the spirit of the relational calculus, which can access Gemstone database schemas. In particular, it can use methods in Gemstone schemas, in much the same way as $L_{alg+\alpha}$ does. Let $\mathcal{F}_{\text{Gemstone}}$ denote the family of Gemstone schemas. Results from [HS89] imply that ($\mathcal{F}_{\text{Gemstone}}$, OPAL) has expressive power greater than $\mathcal{C}$, i.e., there are queries expressible in ($\mathcal{F}_{\text{Gemstone}}$, OPAL) which are not computable. (For example, this language can specify the compliment of the halting problem.) In fact, ($\mathcal{F}_{\text{Gemstone}}$, OPAL) can express exactly the set of mappings computable by Turing machines with (possibly recursive) oracles [HU79]. Of course, in practice OPAL is given a procedural semantics, and so actually has the expressive power of $\mathcal{C}$.

(2) In the Gemstone system attribute values are *untyped*, in the sense that they can hold atoms or sets, where a given set can simultaneously hold atoms and/or tuples and/or sets. (In particular, values with arbitrarily high "set-height" can be formed). Let $\mathcal{F}^{\text{no-new}}_{\text{Gemstone}}$ denote the family of Gemstone schemas with untyped attribute values, with the set-height of types restricted to be 0, and without explicit **new** operations in the methods. Results in [HS89] imply that the OODB language $(\mathcal{F}^{\text{no-new}}_{\text{Gemstone}}, L_{\text{alg}+\alpha})$ is equivalent to $\mathcal{C}$ (restricted to the appropriate family of relational schemas). □

6 Access in Algebraic OODBs

The previous section studied the expressive power and complexity of queries for OODB shcemas in which sets are represented using explicit types in the schema, using either object-based or value-based semantics. Even when the **new** operator is prohibited, queries in this context can have a complexity of PSPACE or higher, which is viewed to be intractable. In this section, we propose an alternative representation of sets based on permitting multi-valued attributes. A query language is proposed in which these attributes can be manipulated using $L_{\text{alg}+\alpha}$, i.e., in a manner reminiscent of the relational algebra. In addition to being tractable, this language provides an approach to set manipulations which may be more convenient than that provided by the languages based on explicit set-valued types.

Essentially, the model of this section is based on the following: (1) prohibiting set types, (2) prohibiting side-effects, (3) introducing multi-valued attributes, and (4) introducing multi-valued methods whose implementations use $L_{\text{alg}+\alpha}$. The resulting language, the *algebraic* OODB access language $\mathcal{L}_{\text{alg-oo}}$, is more expressive than the relational calculus and algebra (Theorem 6.3), and, in fact, contains the class of "generalized transitive closure queries" (see Section 7). In contrast with the OODB schema families studied in the previous section, all queries expressible in $\mathcal{L}_{\text{alg-oo}}$ are in PTIME, hence tractable. It is also shown in Theorem 6.3 that $\mathcal{L}_{\text{alg-oo}}$ is subsumed by CALC+**fixpt** (and, hence, ALG+**while**).

Three sublanguages of $\mathcal{L}_{\text{alg-oo}}$ are also studied. Intuitively, the *single-valued* language allows only schemas with single-valued attributes and single-valued methods; whereas the *multi-valued* language allows multi-valued functions as attributes and methods. Theorem 6.5 states that the single-valued language strictly subsumes the relational algebra ALG but is subsumed properly by the multi-valued language, which is equivalent to $\mathcal{L}_{\text{alg-oo}}$. (We also study a more restricted sublanguage of the single-valued language which is essentially equivalent to ALG.)

We now provide the following definition which is closely related to FDM, except that we do not permit attributes with multiple arguments.

Definition: A *set-free semantic (database) schema* is an ordered 4-tuple $S = (T, ISA, Att_{\text{single}}, Att_{\text{multi}})$ where

1. T is a finite subset of $\mathcal{T}$;

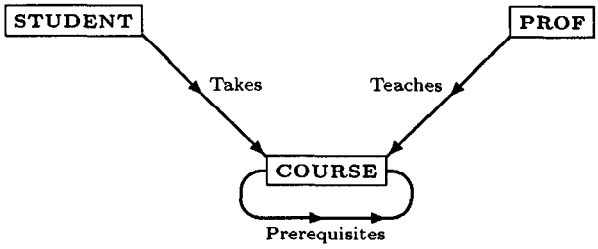

Figure 2: After eliminating ⋆COURSE

2. $ISA \subseteq T \times T$ and (T, ISA) is a directed forest;
3. Att_{single} and Att_{multi} are two functions from $T \to \mathcal{P}^{\text{fin}}(\mathcal{A} \times T)$ satisfying two restrictions as stated in the definition of semantic schema, and the sets of single- and multi-valued attribute names are disjoint. (These specify the *single-valued* and *multi-valued* attributes of S respectively.)

Example 6.1: The semantic schema in Example 3.1 is not a set-free schema. However, since it is of set-height 1, it is easily transformed to a set-free semantic schema. Figure 2 shows part of the schema after elimination of the set type ⋆COURSE. □

Definition: Let $S = (T, ISA, Att_{\text{single}}, Att_{\text{multi}})$ be a set-free semantic schema. An *instance* of S is a triple $d = (O, A_{\text{single}}, A_{\text{multi}})$ where

1. O: $T \to \mathcal{P}^{\text{fin}}(\mathcal{O})$ such that if $s \neq t$ then $O(s)$ and $O(t)$ are disjoint (O^* is defined as before);
2. A_{single} and A_{multi} are two partial functions with domain $\{(o,a) \mid \exists t, t' \in T$, such that $o \in O(t)$ and $(a, t') \in Att_{\text{single}}(t)$ (or $Att_{\text{multi}}(t))\}$. For every such pair (o, a) with associated t, t':
   (a) $A_{\text{single}}(o,a) \in O^*(t')$ if defined; or
   (b) $A_{\text{multi}}(o,a) \subseteq O^*(t')$ if defined.

Note that the above definitions have two main differences compared with the definitions given in Section 3. One is that a set-free schema does not contain set types. On the other hand, multi(or set)-valued attributes are added into the schema. In the relational simulation, the multi-valued attributes are considered as an ordinary binary relation not necessarily satisfying functional dependencies, i.e., if $a \in Att_{\text{multi}}(t)$ then $R_{t,a}$ does not necessarily satisfy any functional dependencies.

Next, we introduce several definitions on the OODB aspect. We start with the notion of *algebraic object-oriented implementation.*

Definition: An implementation is *algebraic object-oriented* if it has the following form:

par: $u_1, ..., u_n$; **var:** $x, y, ..., z; X, Y, ..., Z$;
$s_1; ...; s_k$; **return**(χ)

where

1. $n \geq 0$, $m \geq 1$, $l \in [1..m]$;
2. χ is either X or $\pi_1(X)$; and
3. for $p \in [1..k]$, s_p is (referring to the definition of regular implementation) a **basic operation** statement, a **conditional** statement, or the following **conditional** statement:

   (a) **if** $X\theta\emptyset$ **then** s where s is any nonconditional statement in this list and θ is $=$ or $\neq$,

   or has the following form:
   relational operations:

   (b) $X :=$ {self}

   (c) $X :=$ {self.a} where $a \in \mathcal{A}$ is an attribute

   (d) $X :=$ self.a where $a \in \mathcal{A}$ is an attribute

   (e) $X := \{u_j\}$

   (f) $X :=$ **ext**(t) where $t \in \mathcal{T}$ is a type

   (g) $X := \psi(Y_j)$ where ψ is a unary relational algebra operator

   (h) $X := Y\ \psi\ Z$ where ψ is a binary relational algebra operator

   (i) $X := \alpha(m, Y)$ where $m \in \mathcal{M}$ is a method name

Intuitively, an algebraic object-oriented implementation contains a sequence of statements. Each statement is either an assignment of an individual object which may access attributes, a conditional statement, or a relational operation. Finally, it returns a single object or a set of objects, which determines whether the method is single- or multi-valued.

Example 6.2: Again we consider the method `AllPrereq` in Examples 3.3. Using relational operations, it has the following simple form:

```
AllPrereq'':  var: w, x, y;
              x := self.Prerequisites;
              w := x − x; (empty set)
              if x ≠ ∅ then
                y := α(Allprereq'', x);
              w := π2(y);
              w := w ∪ x;
              return(w)
```

□

Definition: An *algebraic OODB schema* is an ordered 6-tuple $\widehat{S} = (T, ISA, Att_{\text{single}}, Att_{\text{multi}}, Meth, Impl)$ where

1. $S = (T, ISA, Att_{\text{single}}, Att_{\text{multi}})$ is a set-free semantic schema;
2. $Meth : T \rightarrow \mathcal{P}^{\text{fin}}(\mathcal{M})$, such that: if s *ISA* t and $m \in Meth(t)$, then $m \in Meth(s)$;
3. $Impl$ is a partial function from $\{\ (m,t)\ \mid\ m \in Meth(t)\ \} \rightarrow \mathcal{I}$, such that:

   (a) if $m \in Meth(s)$ then there is some $t \succeq s$ such that $Impl(m,t)$ is defined; and

   (b) $Impl(m,t)$ is algebraic object-oriented for each pair (m,t) where $Impl$ is defined.

$Impl^*$ is defined to be a total function on $\{(m,t) \mid m \in Meth(t)\}$ the same as before.

As with regular OODB schemas, algebraic OODB schemas are assumed to satisfy some natural, well-formed restrictions, detailed in the full paper.

In this case, $\widehat{S}$ is also called a *algebraic OODB extension* of S, and S is called the *semantic schema* of $\widehat{S}$. An *instance* of $\widehat{S}$ is an instance of its semantic schema S. Finally, the family of algebraic OODB schemas is denoted as $\mathcal{F}_{\text{alg-oo}}$.

Definition: The *algebraic OODB access language* is $\mathcal{L}_{\text{alg-oo}} = (\mathcal{F}_{\text{alg-oo}}, L_{\text{alg}+\alpha})$.

Since the query language $L_{\text{alg}+\alpha}$ has the full power of the relational algebra, $\mathcal{L}_{\text{alg-oo}}$ is certainly at least as powerful as ALG. Secondly, recursive method calls are allowed in implementations. This suggests that the above subsumption is proper. The following theorem states that $\mathcal{L}_{\text{alg-oo}}$ stands between relational queries and fixpoint queries.

Theorem 6.3: ALG $\sqsubset \mathcal{L}_{\text{alg-oo}} \sqsubseteq$ CALC+**fixpt** ($\sqsubseteq$ ALG+**while**).

The proof uses the algebra + pointwise recursion which will be introduced and discussed in the next section. It remains open whether $\mathcal{L}_{\text{alg-oo}} \equiv$ CALC+**fixpt** or not. Since CALC+**fixpt** is in PTIME, we have:

Corollary 6.4: Every query in $\mathcal{L}_{\text{alg-oo}}$ is in PTIME. □

In the rest of this section, we focus on three sublanguages of $\mathcal{L}_{\text{alg-oo}}$. They all use the same query language $L_{\text{alg}+\alpha}$. The main difference lies in the method definition languages.

Definition: An implementation is *pure single-valued* if it is algebraic object-oriented and contains only **basic operations** and **conditionals**. An implementation is *stable single-valued* if it is single-valued and does not contain any **conditionals**. An implementation is *pure multi-valued* if it is algebraic object-oriented and contains only **relational operations** and **conditionals**.

Definition: Let $\widehat{S} = (T, ISA, Att_{\text{single}}, Att_{\text{multi}}, Meth, Impl)$ be an algebraic OODB schema. $\widehat{S}$ is *single-valued* each implementation in $\widehat{S}$ is pure single-valued. $\widehat{S}$ is *stable single-valued* if

1. each implementation in $\widehat{S}$ is stable single-valued;
2. if $Impl(m,t)$ is defined, $s \preceq t$, and $s \neq t$, then either $Impl(m,s)$ is undefined or is equal to $Impl(m,t)$.

$\widehat{S}$ is *multi-valued* each implementation in $\widehat{S}$ is pure multi-valued.

The family of all single-valued (stable single-valued, multi-valued) schemas is denoted by $\mathcal{F}^{\text{sv}}_{\text{alg-oo}}$ ($\mathcal{F}^{\text{stable}}_{\text{alg-oo}}$, $\mathcal{F}^{\text{mv}}_{\text{alg-oo}}$). Now we denote $\mathcal{L}^{\text{sv}}_{\text{alg-oo}} = (\mathcal{F}^{\text{sv}}_{\text{alg-oo}}, L_{\text{alg}+\alpha})$, $\mathcal{L}^{\text{stable}}_{\text{alg-oo}} = (\mathcal{F}^{\text{stable}}_{\text{alg-oo}}, L_{\text{alg}+\alpha})$, and $\mathcal{L}^{\text{mv}}_{\text{alg-oo}} = (\mathcal{F}^{\text{mv}}_{\text{alg-oo}}, L_{\text{alg}+\alpha})$.

Intuitively, a pure multi-valued OODB schema allows only relational-at-a-time statements in its implementations; a pure single-valued one allows only object-at-a-time statements; and a stable single-valued schema further requires no method overwriting (refinement) and no conditionals.

Theorem 6.5: $\text{ALG} \equiv \mathcal{L}^{\text{stable}}_{\text{alg-oo}} \sqsubset \mathcal{L}^{\text{sv}}_{\text{alg-oo}} \sqsubset \mathcal{L}^{\text{mv}}_{\text{alg-oo}} \equiv \mathcal{L}_{\text{alg-oo}}$.

7 Algebra + Pointwise Recursion

In this section we introduce the language "algebra + pointwise recursion" (ALG+pwrec). This language is of interest for at least three reasons: **(1)** It is a proper extension of the relational algebra (and calculus) which permits some forms of transitivity and recursion, but appears to be more restricted, and thus perhaps easier to implement efficiently, than DATALOG. **(2)** It appears to be closely related to the "traversal recursion" queries of [RHDM86, DS86]. Unlike traversal recursion, the language ALG+pwrec is defined formally and without reference to data structures outside of the relational model. **(3)** It is equivalent in expressive power to the algebraic object-oriented query language $\mathcal{L}_{\text{alg-oo}}$ introduced in the previous section (Theorem 7.1). In particular, it is subsumed by CALC+**fixpt**. (It thus remains open whether ALG+pwrec is properly subsumed by CALC+**fixpt**.)

We begin by defining the language.

Definition: The language *algebra plus pointwise recursion* (ALG+pwrec) consists of queries (programs) Q which have the form

$$s_1, \ldots, s_k, \text{ANS} := Z;\ F_1, \ldots, F_l$$

where

1. for each $j \in [1..l]$, F_j has the form

   func: $f_j(x_1, \ldots, x_{j_{\text{in}}}; X : a)$
   var: $y_1, \ldots, y_{j_{\text{int-atom}}}; Y_1 : a_1, \ldots, Y_{j_{\text{int-rel}}} : a_{j_{\text{int-rel}}}$
   begin $t_1; \ldots; t_{k_j}$ **end**

   where

   (a) f_j is a distinct *function name*, with *input parameters* $x_1, \ldots, x_{j_{\text{in}}}$ (which have *atomic* type); *output parameter* X, which has *relational* type of arity a; *internal atomic parameters* $y_1, \ldots, y_{j_{\text{int-atom}}}$; and *internal relational parameters* $Y_1, \ldots, Y_{j_{\text{int-rel}}}$, where Y_h has arity a_h for $h \in [1..j_{\text{int-rel}}]$.

   (b) for each $i \in [1..k_j]$, t_i is an assignment with one of the following forms (where u's range over atomic variables and V's range over relational variables; and assuming that the appropriate typing restrictions are satisfied)

   i. $u := u_1$
   ii. $V := \{u\}$
   iii. $V := R$; where R is a relation name in the input schema
   iv. $V := \psi(V_1)$; where ψ is a unary relational algebra operator
   v. $V := V_1\ \psi\ V_2$; where ψ is a binary relational algebra operator
   vi. $V := \alpha(f_r, V_1)$; where $r \in [1..l]$)
   vii. **if** $V = \emptyset$ **then** t' **else** t''; where t' and t'' have the form of any other assignment in this list

   (c) for each $i \in [1..k]$, s_i is an assignment, which has any of the forms of items (iv) through (vii) in the previous list (again with V's ranging over relational variables, and satisfying the appropriate type restrictions).

The semantics of $\alpha(f_r, V_1)$ is defined in analogy to the semantics of pointwise method application in $L_{\text{alg+}\alpha}$. On an input schema $\{R_1, \ldots, R_n\}$ and instance d, then *answer* of Q on d, denoted $Q[d]$, is

- ?; if the execution of Q on d (defined in the natural manner) does not terminate, and
- the value assigned to ANS at the completion of the the program; if it does terminate.

As shown in the full paper:

Theorem 7.1: On set-free semantic schemas, $\mathcal{L}_{\text{alg-oo}} \equiv$ ALG+pwrec.

Theorem 7.2: ALG+pwrec $\sqsubseteq$ CALC+**fixpt**, and hence, each query in ALG+pwrec is in PTIME.

Proof: (Sketch) The proof is accomplished by simulating the behavior of an ALG+**pwrec** program P using a single relation R and the fixpoint operator. Suppose that during the execution of P a recursive function call $f(\vec{c})$ occurs. If f hasn't been called on $\vec{c}$ previously, then a new set of tuples is included into R, which will "keep track of" the progress of the computation of f on $\vec{c}$; if f has already been called on $\vec{c}$ then nothing new is inserted into R. Some coordinate values of the newly inserted tuples will have the intuitive meaning "don't know yet". Also, each time that the execution of P actually realizes a value of a function, a coordinate value "don't know yet" in R will be replaced by that value. Execution continues until there are no more "don't know yet" values, or until some steady state is detected. In either case, the entire simulation can be accomplished by a "terminating" computation in CALC+**fixpt**. □

We conclude the section by briefly comparing ALG+pwrec with the family of *traversal recursion queries* introduced in [RHDM86]. These can be viewed as a generalization of transitive closure, and make it possible to perform most common graph traversal algorithms within the context of a database query language. [RHDM86] discusses how traversal recursion queries can be incorporated into a DAPLEX-style language; and it is

clear that they can be incorporated into the relational algebra as well. As discussed in [RHDM86], these extended database languages can be implemented more efficiently than general recursive languages such as DATALOG.

Although we do not have a formal result, it appears that ALG+pwrec properly subsumes the power of the algebra augmented with traversal recursions. As informal evidence, we state the following:

Remark 7.3: Suppose that ALG+pwrec is extended to include arithmetic operators. Then each "generalized transitive closure" query can be expressed using ALG+pwrec. □

References

[AH87a] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[AH87b] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proc. Conf on OOPSLA*, pages 430–440, 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.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. Query languages for object-oriented database systems. In *2nd Intl. Workshop on Database Programming Languages*, June 1989. to appear.

[BCG+87] J Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou. Data model issues for object-oriented applications. *ACM Trans. on Office Information Systems*, 5(1):3–26, 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.

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

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[HS88a] R. Hull and J. Su. On the expressive power of database queries with intermediate types. Technical Report 88-53, Computer Science Department , Univ. of Southern California, 1988. Invited to special issue of *Journal of Computer and System Sciences.*

[HS88b] 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.

[HS89] R. Hull and J. Su. Untyped sets, invention, and computable queries. In *Proc. ACM Symp. on Principles of Database Systems*, 1989.

[HTY88] R. Hull, K. Tanaka, and M. Yoshikawa. Behavior analysis of object-oriented databases: Method structure, exeuction trees, and reachability (extended abstract). Technical Report 88-52, Computer Science Department, University of Southern California, 1988. to appear, *Third Intl. Conf. on Foundations of Data Organization and Algorithms*, Paris, June 1989.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[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.

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

[KV88] G. M. Kuper and M. Y. Vardi. On the complexity of queries in the logical data model. In *Proc. Int. Conf. on Database Theory*, pages 267–280, 1988.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O^2: An object-oriented formal data model. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 424–433, Chicago, June 1988.

[PSM87] A. Purdy, B. Schuchardt, and D. Maier. Integrating an object server with other worlds. *ACM Trans. on Office Information Systems*, 5(1):27–47, 1987.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: A practical approach to supporting recursive applications. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 166–176, 1986.

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

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.