OBJECT IDENTITY AS A QUERY LANGUAGE PRIMITIVE

Serge Abiteboul*
I.N.R.I.A.

Paris C. Kanellakis†
I.N.R.I.A. / Altaïr

Abstract

We demonstrate the power of object identities (oid's) as a database query language primitive. We develop an object-based data model, whose structural part generalizes most of the known complex-object data models: cyclicity is allowed in both its schemas and instances. Our main contribution is the operational part of the data model, the query language IQL, which uses oid's for three critical purposes: (1) to represent data-structures with sharing and cycles, (2) to manipulate sets and (3) to express any computable database query. IQL can be statically type checked, can be evaluated bottom-up and naturally generalizes most popular rule-based database languages. The model can also be extended to incorporate type inheritance, without changes to IQL. Finally, we investigate an analogous value-based data model, whose structural part is founded on regular infinite trees and whose operational part is IQL.

1 Introduction

"Is object relations theory simply a new name for what classical theorists have been doing all along, or is it a fundamentally new system, or an excursion into new realms wholly compatible with classical theory?"
From the cover of Object Relations in Psychoanalytic Theory, *J.R. Greenberg, S.A. Mitchell, Harvard Press 1983.*

Object-oriented database systems will, most probably, be the next generation of commercial database systems (see [10]). They are currently the focus of a great deal of experimentation and research, e.g., [41,46,14,23,17,11]. These recent developments in databases are largely based on concepts and software tools from object-oriented programming, e.g., [25,10,32]. More generally, the integration of programming languages and database systems is an important research activity, for a detailed exposition of the state-of-the-art see [9].

Unfortunately, much of the terminology currently used in object-oriented database systems is overloaded and experts disagree on the precise meaning of concepts such as: "object-identity, types, inheritance, methods and encapsulation etc." Consequently, there has been very little progress on understanding the "principles of object-oriented databases". This is in marked contrast with the previous generation of database systems, where the relational model of [19] provided the basis for many successful implementation efforts and, at the same time, for the development of an elegant and relevant theory [44,29].

An attempt is made in this paper to clarify some of the foundations of object-oriented databases, by showing that they are "an excursion into new realms wholly compatible with classical theory". In particular, we demonstrate that the concept of *object identity* (*oid*) is a powerful programming primitive for database query languages by: *having oid's as the centerpiece of a data model with a rich type system, inheritance and a powerful query language*, called Identity Query Language (IQL).

Oid's have been part of many data models, for example they are called *surrogates* in [20], *l-values* in [37], or *object identifiers* in [4]. They have recently been highlighted as an essential part of object-oriented database systems [30]. A variety of reasons have been given for their use, e.g., structure sharing, updates [4] or the encoding of cyclicity [37]. We use oid's for the traditional encoding of directed (perhaps cyclic) graphs, but also for the manipulation of sets and for making our query language fully expressive. At an intuitive level, oid's are "typed pointers" and IQL is based on a

*Work supported by the PRC BD3.

†On leave from Brown University. The work of this author was supported by: NSF grant IRI-8617344, ONR grant N00014-83-K-0146 ARPA Order No. 4786, and an Alfred P. Sloan Fellowship.

controlled use of "indirection".

The structural part of the *object-based model* described here is a synthesis of elements that existed in the literature. It generalizes the relational data model [19], most complex-object data models, e.g., [1,24,28,34,42,45], and the logical data model (LDM) [37,36]. It can be viewed as the common upper bound of the models used in [37,1]. The pleasant surprise is that little mathematical simplicity had to be traded-off in order to achieve this synthesis. The actual definitions are not much longer than those for the relational model.

The operational part of the data model, the language IQL, is also surprisingly simple both in syntax and semantics. It has three basic properties: (1) it is a rule-based, (2) it can be statically type checked, and (3) it is complete, in the sense that it expresses exactly all database transformations with certain desirable properties. Let us comment on these three points: (1) highlights the declarative nature and mathematical clarity of the programming paradigm used, (2) illustrates what is controlled about the use of pointers, and (3) involves generalizing the basic theorem of [18] from the relational model to a data model with first-order and recursive types.

As in the relational model, there is a clear separation of the notions of instance and schema. As a consequence, the typing of IQL is similar with that of query languages in [37,1,2] and corresponds to strong typing in programming languages. A number of recent language proposals in this area do not have these properties. For example, in [13,40,31] there is no instance-schema separation and the query languages can be viewed as untyped extensions of Prolog.

We give brief overviews (by example) of the structural and operational parts of our data model. The detailed definitions are in Sections 2 and 3, respectively.

Structural Part: An *instance* consists of "data" in the form of: (1) a finite set of *o-values*, i.e., values containing oid's, and (2) a *partial function* ν of oid's to o-values, this mapping is the essence of the data model. Oid's and constants are o-values, but so are finite trees built-out of constants and oid's via finite tuple or set constructors. We allow ν to be partial; this is in order to model incomplete information and will be very useful in the operational part of the model. Note that, repeated applications of ν on oid's give us *pure values*, that are regular infinite trees.

A *schema* contains the information on the "structure of the data" allowed in an instance. In current terminology, it contains the names and types of "persistent data". We have chosen to include two forms of information: (1) *relation names* R for naming *relations*, finite sets of o-values of the same type $\mathbf{T}(R)$, and (2) *class names* P for naming *classes*, finite sets of oid's, where these oid's are mapped through ν to o-values of the same type $\mathbf{T}(P)$. An important assumption is that the classes of any legal instance are pairwise disjoint sets of oid's.

The type language and interpretation is presented in a somewhat nonstandard fashion for the recursive case (i.e., without a μ constructor). The subtle point is that: the recursion is captured by having the types $\mathbf{T}(R)$ and $\mathbf{T}(P)$ refer to base domains or class names.

The dichotomy between relations and classes is the only design decision that slightly complicates the structural part. Its justification is that it greatly simplifies the operational part. Since relations are sets of o-values, duplicates are eliminated from them at a logical level. Thus, it is possible to program directly in popular rule-based formalisms, e.g., Datalog. Relations can name subsets of classes and function as useful temporaries. Also, this distinction allows a direct generalization of both [37] and [1].

Example 1.1 (From *Genesis* 4 and 5.) Schema S has class names *1st-gen* (for first generation), *2nd-gen* and relation names *founded-lineage*, *ancestor-of-celebrity*. Their types are defined as follows:

$\mathbf{T}$(*1st-gen*) = [name: *string*, spouse: *1st-gen*, children: {*2nd-gen*}]

$\mathbf{T}$(*2nd-gen*) = [name: *string*, occupations: {*string*}]

$\mathbf{T}$(*founded-lineage*) = *2nd-gen*

$\mathbf{T}$(*ancestor-of-celebrity*) = [anc: *2nd-gen*, desc: (*string* $\vee$ [spouse: *string*])]

Note that the types can refer to base domain *string* and to class names, but not to relation names. Also, note the cyclicity in the type associated with *1st-gen* and the presence of union types.

Now let us come to an instance I of S. To each relation name R, the instance associates a finite set $\rho(R)$ of o-values of the right type. So, strictly speaking, the type of $\rho(R)$ is $\{\mathbf{T}(R)\}$. To each class name P, the instance associates a finite set $\pi(P)$ of oid's. Classes are assigned disjoint sets of oid's. Partial function ν assigns o-values to the oid's of the instance. Each one of these oid's has a value of the right type or is undefined. So, again strictly speaking, the type of $\pi(P)$ is $\{P\}$ and the type of $\nu(\pi(P))$ is $\{\mathbf{T}(P)\}$.

In instance I, we denote the oid's as *adam, eve, cain, abel, seth, other*. Note that *adam* is distinct from string Adam. I is cyclic, see this by following the ν mapping of the oid's.

π(*1st-gen*) = { *adam, eve* },
π(*2nd-gen*) = { *cain, abel, seth, other* },
ρ(*founded-lineage*) = { *cain, seth, other* },
ρ(*ancestor-of-celebrity*) =
{ [anc: *seth*, desc: Noah], [anc: *cain*, desc: [spouse: Ada]] }
ν(*adam*) = [name: Adam, spouse: *eve*,
children: { *cain, abel, seth, other* }],
ν(*eve*) = [name: Eve, spouse: *adam*,

children: { *cain, abel, seth, other* }],
$\nu(cain)$ = [name: Cain,
occupations: { Farmer, Nomad, Artisan }],
$\nu(abel)$ = [name: Abel, occupations: { Shepherd }],
$\nu(seth)$ = [name: Seth, occupations: { }]
$\nu(other)$ is undefined.
(*Genesis* is rather vague on this point). □

Operational Part: The design of IQL was greatly influenced by both the COL language of [2], for the manipulation of sets, and the detDL language of [7], for the invention of new oid's. The focus was on adding the minimum to Datalog rules in order to obtain an object-based language, that can express all computable queries.

In summary, IQL is inflationary Datalog with negation [33,7], combined with set/tuple types, invention of new oid's, and a weak form of assignment. Inflationary semantics were chosen because of their simplicity and their generality as a flow of control mechanism. We feel that to get the same expressive power similar kinds of extensions would have to be considered, if an algebraic language or a language based on any other paradigm were chosen instead of rules.

The flexibility of a type system, such as the one used here, allows multiple representations of the same information and translation from one representation to another as illustrated next.

Example 1.2 Let the input schema be just a relation R with $\mathbf{T}(R) = [A_1{:}D, A_2{:}D]$ and the output schema be P with $\mathbf{T}(P) = [A_1{:}D, A_2{:}\{P\}]$. The input instance I represents a directed graph G with nodes in D. The desired query is to transform the input instance I into an output instance J representing the same graph. Note that in this new representation every node is associated with an oid, whose value is the pair with the node name for first component, and the set of successors for second component. Note also that the individual oid's used in the output do not matter only their interrelationships do. Let us examine the computation in IQL in four stages:

During the first stage, we produce (in standard Datalog fashion) the set of node names. We use a relation R_0 with $\mathbf{T}(R_0) = [A_1 : D]$. As a shorthand, we do not list the attributes $A_1, A_2, A_3, \ldots$ in the rules, but think of them as first argument of relation, second argument of relation etc. The following rules are used:

$R_0(x) \longleftarrow R(x, y)$
$R_0(x) \longleftarrow R(y, x)$

In the second stage, we produce two oid's per node using a semantics in the style of detDL [7]. We use a relation R' with $\mathbf{T}(R') = [A_1 : D, A_2 : P, A_3 : P']$ whose tuples contain oid's from class P and from class P'. Class P' is a class with $\mathbf{T}(P') = \{P\}$, i.e., it's oid's have values that are sets of oid's from P. The following rule invents two oid's for each node, one of which will go into class P and the other into class P'.

$R'(x, p, p') \longleftarrow R_o(x)$

Note how the variables p, p' in the head are not in the body. When the new oid's are invented they are placed in the proper classes and they are automatically assigned default values: p is undefined and p' is the empty set (because of the set valued type of P').

In the third stage, we nest the oid's representing nodes in P into sets of successors of a node. This nesting of elements q is done by using the oid's p' of P' as temporary names. Each p' is set valued and its value, noted $\widehat{p'}$, is a set in which the corresponding q's are collected. This dereferencing and assignment to objects in P' simulates the effect of a COL data-function [2] or a grouping in LDL [15].

$\widehat{p'}(q) \longleftarrow R'(x, p, p'), R'(y, q, q'), R(x, y)$

In the final stage, the nodes of P have been grouped into P', and the connection in R' between x, p, p' is used to produce the desired result. Note that the value of some node p is a tuple with the name of the node as first component, and a set of P-oid's as a second component. This weak form of assignment is performed only when $\hat{p}$ was undefined (see [5]). No further changes are made to $\hat{p}$.

$\hat{p} = [x, \widehat{p'}] \longleftarrow R'(x, p, p')$

We have presented the program in four separate stages. We need not separate the stages. It is possible through standard techniques (using negation) to slightly modify the rules above and think of them as operating in parallel with inflationary semantics. A useful construct, definable in IQL, is that of sequential composition (;). In fact, only the last rule need to be modified by separating it with a (;) from the rest of the rules. □

An important primitive in the language is the invention of oid's. This serves a triple goal: (1) objects may be part of the result and oid's must be assigned to them, (2) invented oid's are used for set manipulation, (3) they are also used to obtain completeness in the sense of [18]. The reason we use (1) is to code sharing of structures and cyclic structures. Regarding (2), the rule-based language does not need to have any mechanism such as *grouping* in LDL [15], *data-functions* in COL [2], or *universal quantification* [35]. Thus, one of our contributions is to show that: *the manipulation and creation of sets can be realized only using invented oid's.*

We examine (3) in detail in Section 4. The notion of completeness of [18] is adapted to our context. Intuitively, the language must capture all transformations that are recursively enumerable and that preserve some isomorphism properties [18,26]. A basic contribution is a completeness result for IQL. *For disjoint input-output schemas, we show that all database transformations are expressible in IQL, up to copy elimination.* In many cases we can express copy elimination in IQL, but it is open if this technical restriction is necessary. Dis-

joint input-output schemas are sufficient for the study of queries and updates, such as insertions. To obtain completeness for non-disjoint schemas, we need to add non-inflationary features to IQL. These are based on the study of deletions in [7]. (For other completeness results see[8,7,27,22]).

In Section 5, we specialize IQL using a number of syntactic restrictions. This specialization allows us to discover as IQL *sublanguages most of the popular rule-based formalisms.* We also show that these restrictions can be used to guarantee efficient query evaluation (i.e., with PTIME *data-complexity*) see also [3].

In summary, IQL is both a mathematical model of computation with types and (particularly in its range restricted form IQL^{rr}) a useful high level query language. Like Prolog, it can be used to manipulate unbounded structured terms, but unlike Prolog it is typed, it has negation, it is a good candidate for conventional database optimizations, and its semantics are not complicated by depth-first search strategies.

The subsequent sections of our paper deal with two issues which, we believe are orthogonal to the structural and the operational parts of our object-based model. The first is type inheritance (Section 6) and the second is the relationship of object-based with value-based (Section 7).

Type Inheritance: In all the development of IQL we make crucial use of a technical condition, the pairwise disjointness of the various classes of an instance. This condition guarantees the soundness and the static typability of IQL programs. However, the removal of this condition is necessary if one is to study *type inheritance* as proposed in [16]. With inheritance, the disjointness condition on the classes is replaced by a less restricted condition that, we argue, is natural. We show that, under this limited addition, type inheritance has simple semantics. The union types of our object-based data model are critical in this development. What we observe is that: union types are a more general mechanism for sharing structure than type inheritance. As a result, IQL can be used (at no cost of expressive power) to deal with schemas with inheritance.

Value-Based vs Object-Based: Oid's can be viewed as a syntactic trick to avoid manipulating recursive objects. The same is true for the use of class names in the type syntax. Even with these devices, recursive structures stay in the background in a fundamental way. Object-based systems often allow features such as *equality-by-value*, which is a precise way of addressing the underlying infinite objects. We illustrate a natural connection with a *value-based model* founded on regular infinite trees [21]. Our analysis allows us to show that IQL can serve as a language for this model as well. Object identities, in this context, loose all semantic denotation to become purely, primitives of the language. This is a nontrivial link between value-based and object-based [43]. A value-based point of view can be used to understand *pure-values* (no oid's), *pure types* (no classes in the type syntax) and *equality-by-value* (as a coercion mechanism).

A major motivation for our work was the study of the formal aspects of the O_2 system, [11]. O_2 is a multilanguage object-oriented database system (around,for instance, *C* and *Basic*). Its structural data model, see [39,38], is a subset of the structural data model with inheritance described here. The prototype implementation presented in [11] does not have union types, but does have type inheritance. Inheritance in O_2 is constrained by (pure) type compatibility: this facilitates coercions and the use of inheritance in queries [39]. Some stand-alone query languages, besides CO_2, are also investigated in [12].

Because, of space limitations we have omitted proofs and many important details from this extended abstract; for these see [6].

2 An Object-Based Data Model

We assume the existence of the following countably infinite and pairwise disjoint sets of atomic elements: (1) *relation names* $\{R_1, R_2, \ldots\}$, (2) *class names* $\{P_1, P_2, \ldots\}$, (3) *attributes* $\{A_1, A_2, \ldots\}$, (4) *constants* $D = \{d_1, d_2, \ldots\}$, and (5) *object identities* or *oid's* $O = \{o_1, o_2, \ldots\}$. Throughout our exposition, we use the generic notation $[A_1 : \cdots, \ldots, A_k : \cdots]$ (where k is a nonegative integer) for a *tuple* formed using *any* k *distinct* attributes $A_1, \ldots, A_k$ (when $k > 0$) and for the *empty tuple* [ ] (when $k = 0$). The empty set is denoted $\emptyset$ or $\{\}$.

Definition 2.1 The set of *o-values* is the smallest set containing $D \cup O$ and such that, if $v_1, \ldots, v_k$ $(k \geq 0)$ are o-values then so are: (1) $[A_1 : v_1, \ldots, A_k : v_k]$ and, (2) $\{v_1, \ldots, v_k\}$.

Definition 2.2 Let $\mathbf{R}$ be a finite set of relation names and let $\mathbf{P}$ be a finite set of class names.
(1) An *o-value assignment* for $\mathbf{R}$ is a function ρ mapping each name in $\mathbf{R}$ to a *finite* set of o-values. (2) An *oid assignment* for $\mathbf{P}$ is a function π mapping each name in $\mathbf{P}$ to a *finite* set of oid's and
we call π *disjoint* if $P \neq P'$ implies $\pi(P) \cap \pi(P') = \emptyset$ (where $P, P' \in \mathbf{P}$).

By finiteness, each *relation* $\rho(R)$ $(R \in \mathbf{R})$ and each *class* $\pi(P)$ $(P \in \mathbf{P})$ is itself an o-value. Since o-values are defined using finite tupling and finite subsetting, it is possible to represent them using *finite trees* with base, set and tuple nodes.

The syntax and semantics of types are now defined using a given finite set of class names **P** and an oid assignment π for **P**. The set of *type expressions*, called *types*(**P**), is given by the following abstract syntax, where τ is a type expression, P an element of **P** and $k \geq 0$:

$$\tau = \emptyset \mid D \mid P \mid [A_1 : \tau, \ldots, A_k : \tau] \mid \{\tau\} \mid (\tau \vee \tau) \mid (\tau \wedge \tau)$$

For an oid assignment π, each type expression τ is given a set of o-values as it's *interpretation* $[\![\tau]\!]_\pi$, in the following natural fashion:

- $[\![\emptyset]\!]_\pi = \emptyset$, $[\![D]\!]_\pi = D$,
  $[\![P]\!]_\pi = \pi(P)$ (for each $P \in \mathbf{P}$),
- $[\![(\tau_1 \vee \tau_2)]\!]_\pi = [\![\tau_1]\!]_\pi \cup [\![\tau_2]\!]_\pi$ and
  $[\![(\tau_1 \wedge \tau_2)]\!]_\pi = [\![\tau_1]\!]_\pi \cap [\![\tau_2]\!]_\pi$,
- $[\![\{\tau\}]\!]_\pi =$
  $\{\{v_1, \ldots, v_j\} \mid j \geq 0$, and $v_i \in [\![\tau]\!]_\pi$, $i = 1,\ldots,j\}$,
- $[\![[A_1 : \tau_1, \ldots, A_k : \tau_k]]\!]_\pi =$
  $\{[A_1 : v_1, \ldots, A_k : v_k] \mid v_i \in [\![\tau_i]\!]_\pi$, $i = 1,\ldots,k\}$.

We sometimes represent a type expression τ by its *parse tree*, which has internal nodes labeled by tupling ($\times$), finite set construction ($\star$), union ($\vee$), intersection ($\wedge$). We say that:

- τ is *intersection reduced* if in τ's parse tree, no $\wedge$-node is an ancestor of a $\times, \star$, or $\vee$-node,
- τ is *intersection free* if τ's parse tree has no $\wedge$-node.

Two type expressions τ_1, τ_2 are *equivalent* (*over disjoint oid assignments*), if for each (disjoint) oid assignment π they have the same interpretations. By easy algebraic manipulation, we have:

Proposition 2.1 For each type expression (1) there is an intersection reduced, equivalent type expression, (2) there is an intersection free, equivalent over disjoint oid assignments type expression.

Most of our analysis uses disjoint oid assignments and therefore, by this proposition, intersection can be eliminated. After these preliminaries, we present schemas and instances and comment on their definitions.

Definition 2.3 A schema S is a triple (**R**, **P**, **T**), where **R** is a finite set of relation names, **P** is a finite set of class names, and **T** is a function from **R**$\cup$**P** to *types*(**P**).

Definition 2.4 An instance I of schema **(R, P, T)** is a triple (ρ, π, ν), where ρ is an o-value assignment for **R**, π is a disjoint oid assignment for **P**, and ν is a partial function from the set of oid's $\cup$ { $\pi(P) \mid P \in \mathbf{P}$ } to o-values, such that:

1. $\rho(R) \subseteq [\![\mathbf{T}(R)]\!]_\pi$, for each $R \in \mathbf{R}$,
2. $\nu(\pi(P)) \subseteq [\![\mathbf{T}(P)]\!]_\pi$, for each $P \in \mathbf{P}$,
3. ν is total on $\pi(P)$, for each $P \in \mathbf{P}$ with $\mathbf{T}(P) = \{\tau\}$.

It is important to note that: each oid occurring in I (i.e., in the ranges of ρ, π, ν) must belong to some $\pi(P)$ (where $P \in \mathbf{P}$). This easily follows from conditions (1) and (2) of the instance definition and from the semantics of types.

Let I=(ρ, π, ν) be an instance of a schema S=(**R**, **P**, **T**). A *set valued* oid in I is an oid belonging to a class P, where $\mathbf{T}(P)$=$\{\tau\}$ for some τ. Since an oid can only belong to one class, this is a well defined notion. The information contained in I can be represented in a "logic programming" notation as follows:

$$\begin{aligned} \textit{ground-facts(I)} = {} & \{R(v) \mid v \in \rho(R), R \in \mathbf{R}\} \cup \\ & \{P(o) \mid o \in \pi(P), P \in \mathbf{P}\} \cup \\ & \{\hat{o}(v) \mid v \in \nu(o), o \text{ set valued}\} \cup \\ & \{\hat{o} = v \mid v = \nu(o), o \text{ non-set valued}\}. \end{aligned}$$

It is easy to see that *ground-facts(I)* is an alternative representation of I. Recall condition (3) in Definition 2.4 that ν must be total for set valued oid's. Based on this, we follow the convention that: if for some set valued oid o there is no ground fact $\hat{o}(v)$ then $\nu(o) = \{\}$, and if for some non-set valued oid o there is no ground fact $\hat{o} = v$ then ν is undefined at o.

Finally, we use the terminology: *instances(S)* for the set of all instances of schema S; *objects(I)* for the set of all oid's occurring in I; *constants(I)* for the set of all constants occurring in I.

Remark: The structural part of the model generalizes that of many previously introduced data models, e.g., the relational and most complex-object data models as well as LDM. In addition, incomplete information can be modeled using oid's with undefined value. Besides this, there is an important technical reason for having oid's with undefined values. The language IQL builds objects in stages, and oid's with undefined values are used in the intermediate stages. For o-values of set type $\{\tau\}$, we use $\{\}$ in order to achieve the same effect without any ambiguity.

3 The Identity Query Language

We first need to define projections of schemas and instances, in order to describe the inputs and outputs of

programs. A schema $S' = (\mathbf{R}',\mathbf{P}',\mathbf{T}')$ is the *projection* of schema $S = (\mathbf{R},\mathbf{P},\mathbf{T})$ if we have $\mathbf{R}' \subseteq \mathbf{R}$, $\mathbf{P}' \subseteq \mathbf{P}$, and $\mathbf{T}'$ is the mapping $\mathbf{T}$ on $\mathbf{R}' \cup \mathbf{P}'$. Given an instance I of S, its *projection* on S', denoted $I[S']$, is defined in the obvious way and is an instance of S'.

An Identity Query Language (IQL) program $\Gamma(S, S_{in}, S_{out})$ consists of rules over schema S and expresses a binary relation on instances. This relation is between instances over the *input schema* S_{in} and instances over the *output schema* S_{out}, where S_{in}, S_{out} are two projections of S. Intuitively, the input to a program is an instance I over S_{in}, the computation of the program defines an instance J over S, and the output is $J[S_{out}]$.

3.1 Syntax

The syntax for a program $\Gamma(S, S_{in}, S_{out})$ is a finite set of rules over $S = (\mathbf{R},\mathbf{P},\mathbf{T})$, where terms, literals and rules are defined as follows.

Terms: Assume that there are pairwise disjoint, countably infinite sets of variables for each τ in *types*($\mathbf{P}$) and that $k \geq 0$. The *terms* and their types are:

- each variable x of type τ is a term of type τ,
- each R in $\mathbf{R}$ is a term of type $\{\mathbf{T}(R)\}$ and each P in $\mathbf{P}$ is a term of type $\{P\}$,
- for each P in $\mathbf{P}$ and variable x of type P, $\hat{x}$ is a term of type $\mathbf{T}(P)$,
- for $t_1,...,t_k$ terms of type τ, $\{t_1,...,t_k\}$ is a term of type $\{\tau\}$,
- for $t_1,...,t_k$ terms of types $\tau_1,...\tau_k$, $[A_1{:}t_1,...,A_k{:}t_k]$ is a term of type $[A_1{:}\tau_1,...,A_k{:}\tau_k]$.

Literals: Let t_1, t_2 be terms, then $t_1{=}t_2$, $t_1(t_2)$ are *positive literals*, and $t_1{\neq}t_2$, $\neg\, t_1(t_2)$ are *negative literals*. A *literal* (positive or negative) is *typed* when:

- for literals $t_1(t_2)$ or $\neg\, t_1(t_2)$, the term t_1 is of type $\{\tau\}$ and the term t_2 of type τ,
- for literals $t_1{=}t_2$, $t_1{\neq}t_2$, the terms t_1 and t_2 are both of type τ.

A *fact* is any typed positive literal of the following forms:

- $R(t)$ for R in $\mathbf{R}$, $P(t)$ for P in $\mathbf{P}$,
- $\hat{x}(t)$ where $\hat{x}$ is of set type, or $\hat{x}{=}t$ where $\hat{x}$ is not of set type.

Rules: A *rule* r is an expression of the form $L \longleftarrow L_1,...,L_k$ $(k \geq 0)$, where L is a literal called *head*(r) and $L_1,...,L_k$ is a set of literals called *body*(r) and:

1. *head*(r) is a fact and thus is typed,
2. all literals in *body*(r) are typed; except for $t_1 = t_2$ with t_1 of type τ and t_2 of type $\tau \vee \tau'$,
3. each variable occurring in *head*(r) and not in *body*(r) has type P for some P in $\mathbf{P}$.

Remark: Terms, literals and rules as defined here are pretty much standard, with some important additions. These are: (1) the typing for $R, P, \hat{x}$ in terms, (2) the relationship of the syntax of heads or facts with the ground facts of an instance, (3) the more liberal typing of equality in the bodies that will be used to deal with the union of types, and (4) the type restriction for variables in the heads and not in the bodies. Finally, note that we have not included among the terms any constants for the elements of D. This is in order to study a "pure" language as in [18,7]. Constants can be added easily without changing the framework.

3.2 Semantics

The semantics of program $\Gamma(S, S_{in}, S_{out})$ is a binary relation $\gamma = \gamma(\Gamma)$ on instances. The pair (I,I') is in γ if: I is in *instances*(S_{in}), I' is in *instances*(S_{out}), and $I' = J[S_{out}]$ for some J in *instances*(S) where (I,J) is in the program's *inflationary fixpoint operator* γ_∞.

We now formally define the inflationary fixpoint operator of a program using valuations, satisfaction, and the one step inflationary operator. These notions are straightforward extensions of those used for the semantics of detDL in [7]. They are slightly complicated by two aspects of the language: (1) the particular mechanism used for oid invention, and (2) the *weak* assignment of o-values to non-set valued oid's based on condition (*) below.

Valuations: Given an instance $I{=}(\rho,\pi,\nu)$, a valuation θ is a partial function from variables to o-values such that: if θx is defined and x is of type τ, then (1) θx is in τ's interpretation given π, and (2) the constants occurring in θx are from *constants*(I). A valuation (given I) can be extended to terms t as θt defined below. Note that θ is a partial mapping on variables, so θt may be undefined for some variables and some terms.

- $\theta R = \{v \mid R(v)$ in *ground-facts*(I)$\}$ and $\theta P = \{o \mid P(o)$ in *ground-facts*(I)$\}$,
- $\theta\hat{x} = \{\, v \mid$ ground fact $\widehat{\theta x}(v)$ is in *ground-facts*(I)$\}$ where $\hat{x}$ is of set type,

- $\theta\widehat{x} = v$ if ground fact $\widehat{\theta x} = v$ is in *ground-facts*(I) where $\widehat{x}$ is not of set type,
- $\theta\{t_1,...,t_k\} = \{\ \theta t_1,...,\theta t_k\}$, $\theta[A_1{:}t_1,...,A_k{:}t_k] = [A_1{:}\theta t_1,...,A_k{:}\theta t_k]$ ($k \geq 0$).

Satisfaction and valuation-domain: Let I be an instance and θ a valuation (given I) that must be defined on terms t_1,t_2. We say that (1) $I \models \theta[t_1(t_2)]$ if $\theta t_2 \in \theta t_1$, (2) $I \models \theta[t_1{=}t_2]$ if $\theta t_1 = \theta t_2$, (3) $I \models \neg\theta[t_1(t_2)]$ if $\theta t_2 \notin \theta t_1$, and (4) $I \models \theta[t_1{\neq}t_2]$ if $\theta t_1 \neq \theta t_2$. In addition, let r be a rule. We say that $I \models$ *body*(r) if I *satisfies* ($\models$) all the literals in *body*(r).

Given a program Γ and an instance I, the *valuation-domain*, denoted *val-dom*(Γ,I), is defined as follows:
val-dom(Γ,I) = { (r,θ) | $r \in \Gamma$, $I \models \theta$*body*(r),
θ is a valuation exactly on variables in *body*(r),
and there is *no* extension $\bar{\theta}$ of θ such that
$I \models \bar{\theta}$ *head*(r)}

By the extension $\bar{\theta}$ of θ, we mean a valuation (given I) that agrees with θ on the variables occurring in *body*(r) and that is also defined on the variables occurring in *head*(r) but not in *body*(r).

The significance of the valuation-domain is that if one thinks of I as the "current state", then each (r,θ) contributes to augmenting I. Thus, the valuation-domain is the set of valuations that participate in the derivation of new ground facts. New ground facts can be added to I either using old oid's and constants, or inventing new oid's. Here are the laws governing the invention of oid's:

Invention and valuation-map: A *valuation-map* η, for program Γ and instance I, is a function defined on *val-dom*(Γ,I) with the following properties. For each (r,θ) we have that $\eta(r,\theta)$ is a valuation of the variables in r such that:

- if x in *body*(r) then $\eta(r,\theta)x = \theta x$
  (i.e., $\eta(r,\theta)$ is an extension of θ),
- if x in *head*(r) and not in *body*(r) then $\eta(r,\theta)x$ is in (O − *objects*(I))
  (i.e., $\eta(r,\theta)x$ is new, recall that x has type P for some P in **P**),
- if x in *head*(r) and not in *body*(r), and x' in *head*(r') and not in *body*(r')
  then $r \neq r'$ or $\theta \neq \theta'$ or $x \neq x'$ implies $\eta(r,\theta)x \neq \eta(r',\theta')x'$
  (i.e., all inventions happen in parallel, producing distinct oid's for each parallel branch).

Inflationary operators: Given program Γ, the *inflationary one-step operator* γ_1 is a binary relation on instances. The pair of instances (I,J) is in γ_1 if there exists a valuation-map η for Γ and I and:
ground-facts(J) = *ground-facts*(I) ∪
{ $\eta(r,\theta)$*head*(r) | for some r,θ subject to (*) below }
∪ { $P(o)$ | for some r,θ and x of type P,
$o = \eta(r,\theta)x$ and o is invented, }

(*) Let o be non-set valued. If $\widehat{o}$ is undefined in I and a single new ground fact $\widehat{o}{=}v$ is derived, then it is added to *ground-facts*(J). If $\widehat{o}$ is defined in I or if two distinct new facts $\widehat{o}{=}v$ and $\widehat{o}{=}v'$ are derived, then the new derived ground facts about $\widehat{o}$ are ignored.

Given program Γ, its *inflationary fixpoint operator* γ_∞ is a binary relation on instances. The pair of instances (I,J) is in γ_∞ if there exists a finite sequence $I_0{=}I,\ldots,I_i,\ldots,I_n{=}J$ with (1) for all $i > 0$ we have $(I_{i-1},I_i) \in \gamma_1$, and (2) for all J' if $(J,J') \in \gamma_1$ then $J{=}J'$.

A determinate naive inflationary evaluator: It is possible to have nonterminating computations in IQL, see condition (2) right above. Also, because of the quantification over valuation-maps η in the definition of γ_1, the binary relation γ_1 contains (I,J) pairs for all possible legal choices of invented oid's. It follows that, there could be many J associated to a single instance I, as pairs (I,J) in γ. However, as we shall see in Theorem 4.1, all these J are isomorphic to each other. It is easy to define an algorithm for evaluating IQL programs, based on the semantics above. This *naive inflationary evaluator* proceeds in iterations: *in each iteration it determines the valuation-domain and picks a valuation-map; it stops if no ground fact is added.* The output is *independent* of the choice of valuation-map made by this evaluator, up-to renaming of invented oid's.

In IQL, one goal of type checking is to guarantee the soundness of programs. In other words, type checking is used to guarantee that the result is a *correct* instance. Another goal of type checking in IQL is to increase the *efficiency of evaluation*, e.g., to decrease the size and the cost of computing the valuation-domain. This latter use is a major justification for the separate notions of schema and instance in data models: "the schema contains the type information that is used to make retrieval efficient". It is easy to see that IQL programs can be statically type-checked There is one exception. Namely, the treatment of ground facts $\widehat{o}{=}v$ involves some checking during the evaluation; see (*) in the definition of the one-step inflationary operator. However, this exception does not invalidate our claim. The dynamic check performed here is of very small cost and does not entail checking the whole type. We check only if an oid has a value or is undefined. The cost is less than even recording the derived facts. Unfortunately, statically deciding if this inexpensive check will be used in some evaluation

is not recursive; see [5].

3.3 Shorthands and Examples

We accept $R(t_1,...,t_k)$ as an alternative notation for $R([A_1{:}t_1,...,A_k{:}t_k])$, when some implicit ordering on the attributes is understood. It is now clear that each *Datalog* program can be viewed as a valid IQL program on a relational schema, and that its Datalog and IQL semantics are identical. The same applies to *Datalog with negation and inflationary semantics.*

Continuing with relational schemas, other relational languages can be viewed as IQL sublanguages, for example *detDL* [7]. The differences between detDL and IQL restricted to relations are: slightly different semantics for valuation-domains and invented constants in detDL versus invented oid's in IQL. However, it is very simple to simulate detDL in IQL.

It is shown in [7] that control mechanisms such as *composition, if-then-else*, and *while-statements* can be simulated in detDL (using negation and inflationary semantics). These mechanisms can now be used as shorthands. In particular, we use ";" to denote composition. The transformation expressed by an IQL program $\Gamma_1;\Gamma_2$ is the composition of the transformations expressed by Γ_1 and Γ_2. Using composition, it is easy to see that *relational calculus* queries and *Datalog with stratified negation* are expressible in IQL almost verbatim.

Now consider complex-objects. The most famous operations on complex-objects are *nest* and *unnest*. Nest/unnest in IQL resembles the expression of these operations in the language *COL* [2,3]. The next example shows the IQL realization. For greater clarity we use capital letters, e.g., X,Y, for set variables.

Example 3.1 Let **(R,P,T)** be a schema, R_1, R_2, R_3 $\in$ **R**, $\mathbf{T}(R_1)$= $\mathbf{T}(R_3)$= $[A_1{:}D,A_2{:}\{D\}]$, and $\mathbf{T}(R_2)$= $[A_1{:}D,A_2{:}D]$. We want to unnest R_1 into R_2, and then nest R_2 into R_3. For unnesting use single rule:

$R_2(x,y) \longleftarrow R_1(x,Y), Y(y)$.

For nesting, use an auxiliary class P associated with $\mathbf{T}(P) = \{D\}$, and auxiliary relations R_4, R_5 associated with $\mathbf{T}(R_4) = D$, $\mathbf{T}(R_5) = [A_1{:}D,A_2{:}P]$.

Nesting is realized with $\Gamma_1;\Gamma_2$ where Γ_1 is:

$R_4(x) \longleftarrow R_2(x,y)$

$R_5(x,z) \longleftarrow R_4(x)$

$\hat{z}(y) \longleftarrow R_2(x,y), R_5(x,z)$

and Γ_2:

$R_3(x,\hat{z}) \longleftarrow R_5(x,z)$

Γ_1 creates one oid z per x in the A_1-column of R_2. The value of the oid z is the set of values paired to its x in the A_2-column of R_2. The program Γ_2 starts after Γ_1 completes and constructs the result. Note how attributes were omitted from the rules, without any ambiguity. □

It should be noted that no invention is required in COL to perform the nesting: it is realized using *data functions.* A data function can be viewed as a *parametrized relation* and is therefore based on a more elaborate concept than the relations in IQL. However, data functions can be simulated in IQL using invented oid's. We chose here to have oid invention, since such a feature serves many other purposes as well in our context.

One can show that each COL query can be computed using an IQL program. The proof is easy given the above programs for nest/unnest. As a consequence, all *algebraic operations on complex objects* of [24,28,42,1], and the calculus queries of [1,34] are expressible in IQL. Also, it is easy to show that all calculus and algebra queries in LDM can be simulated in IQL.

One important operation found in the algebra for LDM and the algebra for complex-objects of [1] is *powerset.* This operation is expensive: it is exponential in the input size. Indeed, we will emphasize sublanguages of IQL that cannot express the powerset, but can express important classes of queries evaluable in time polynomial in the input instance size. The powerset operation is considered in the next example. This example will provide all the necessary guidelines for the restrictions that will be imposed on IQL to obtain efficiently evaluable sublanguages.

Example 3.2 First suppose that the input consists of a single relation R of type D and the output, of a single relation R_1 of type $\{D\}$. The powerset of R is computed in R_1 by:

$R_1(X) \longleftarrow X = X$

where X is a variable of type $\{D\}$. Indeed, since R is the single input relation, by the definition of valuation (given I) the variable X will range only over the subsets of R, and R_1 will contain the powerset of R.

The obvious problem is that the variable X is not *range-restricted* in the program (see Section 5 for a formalization of range-restriction). However, the powerset can also be computed in a range-restricted manner using oid's. Let R and R_1 be the input and output as above. We also use a class P with type $\{D\}$, and an auxiliary relation R_2 with type $[A_1{:}\{D\}, A_2{:}\{D\}, A_3{:}P]$.

The powerset program has the rules:

$R_1(\{\}) \longleftarrow$

$R_1(\{x\}) \longleftarrow R(x)$

$R_2(X,Y,z) \longleftarrow R_1(X), R_1(Y)$

$\hat{z}(x) \longleftarrow R_2(X,Y,z), X(x)$

$\hat{z}(y) \longleftarrow R_2(X,Y,z), Y(y)$

$R_1(\hat{z}) \longleftarrow P(z)$.

One can check that this computes the powerset in a constructive way. Suppose that relation R is $\{d_1,d_2,d_3\}$. Then $\{\}$, $\{d_1\}$, $\{d_2\}$, $\{d_3\}$ are first obtained then $\{d_1,d_2\}$, $\{d_2,d_3\}$, etc. In this computation some subsets are obviously derived more than once.

Note that in this second way of computing the powerset, invention of oid's occurs in a "loop". Such recursion

with invention of oid's may clearly be the cause of non-terminating computations. For instance, let R_3 be a relation with $\mathbf{T}(R_3) = [A_1{:}P, A_2{:}P]$. Then the rule

$R_3(y,z) \longleftarrow R_3(x,y)$

may cause the non-termination of the computation. □

As illustrated by the graph example of the introduction, IQL also allows the creation of objects and the sharing of objects. We refer to that example for many features of IQL such as: Datalog rules, set manipulation, invention of oid's bounded by a polynomial in the size of the input, composition, and weak assignment to non-set oid's.

The union of types is treated in IQL in a special fashion. This is based on allowing the use of a less constrained typing condition in the rule bodies. This condition (2 in the syntax of rules) can be viewed as equality modulo *coercion*. The following is an example involving union types.

Example 3.3 Consider the two schemas:

S has only one class P with

$\mathbf{T}(P) = P \vee [A_1 : P, A_2 : P]$ and

S' has only one class P' with

$\mathbf{T}'(P') = [A_1 : \{P'\}, A_2 : \{[A_1 : P', A_2 : P']\}]$.

We use one temporary relation R with $\mathbf{T}(R) = [A_1 : P, A_2 : P']$ and omit the attributes, when there is no ambiguity.

Instances over S can be "losslessly" transformed to S' instances using rules:

$R(x, x') \longleftarrow P(x)$

$\widehat{x'} = [\{y'\}, \emptyset] \longleftarrow R(x, x'),\ R(y, y'),\ y = \widehat{x}$

$\widehat{x'} = [\emptyset, \{[y', z']\}] \longleftarrow$

$R(x, x'), R(y, y'), R(z, z'), [y, z] = \widehat{x}.$

An "inverse" mapping from S' to S can be realized using rules:

$R(x, x') \longleftarrow P'(x')$

$\widehat{x} = w \longleftarrow R(x, x'),\ R(y, y'),\ y = w,\ \widehat{x'} = [\{y'\}, \emptyset]$

$\widehat{x} = w \longleftarrow R(x, x'), R(y, y'), R(z, z'),$

$[y, z] = w, \widehat{x'} = [\emptyset, \{[y', z']\}].$

Note the use of coercions in the bodies. For instance, in the the first program, $\widehat{x}$ is of type $P \vee [A_1{:}P, A_2{:}P]$, whereas y, z are of type P. In the second program, w has type $P \vee [A_1{:}P, A_2{:}P]$, different from the types of y and $[y, z]$. We use w in order to have typed heads. □

4 IQL expressibility

We call an isomorphism h on $D \cup O$, that maps D to D and O to O, a *DO-isomorphism*. Clearly, each such isomorphism can be extended to o-values and instances. An *O-isomorphism* is an isomorphism on O. Each O-isomorphism can be viewed as a DO-isomorphism by extending it with: $hd = d$ for each d in D. Thus, an O-isomorphism can be viewed as an isomorphism over o-values and instances.

The following definition states the four conditions that a binary relation on instances should satisfy, in order to qualify as a db-transformation. The first three conditions are standard and capture *well-typedness*, *effective computability*, and *genericity*. The justification for genericity is that a query language should not "interpret" atomic elements, such as constants and oid's, see [18,26]. The fourth condition is new and expresses a form of *functionality*.

Definition 4.1 A binary relation γ on instances is a *db-transformation* if:

1. $\exists\, S, S'$ such that $\gamma \subseteq \mathit{instances}(S) \times \mathit{instances}(S')$,
2. γ is recursively enumerable,
3. h a DO-isomorphism and $(I, J) \in \gamma$ imply that $(hI, hJ) \in \gamma$,
4. $(I, J_1), (I, J_2) \in \gamma$ imply that there exists an O-isomorphism h' such that $h'J_1 = J_2$.

It follows from conditions 3-4 that no new constants can appear in the output. More precisely,

if (I, J) is in a db-transformation γ then

$\mathit{constants}(J) \subseteq \mathit{constants}(I)$.

In contrast, the kind of functionality enforced by condition (4) allows the presence of oid's in the output that were not in the input. This is a significant addition to the frameworks of [7,18]. It is important to be able to create new oid's in the output, if one wishes to manipulate in a general fashion the types available in the data model.

Another intuition formalized by (4) is that the oid's as atomic elements are irrelevant, only their interrelationships matter. Consider the IQL example of the introduction. The oid's of the nodes of the output instance do not matter: "if two instances are O-isomorphic they contain the same information".

We now prove a soundness theorem: that IQL programs define only db-transformations. It follows that IQL programs are determinate in the sense of condition (4). Note that the disjointness of oid assignments is important to guarantee soundness, see [6].

Theorem 4.1 The semantics of an IQL program is a db-transformation.

This soundness theorem raises a natural completeness question: are all db-transformations expressible in IQL? Consider a relation name R common in both input and output schemas. A problem is that the inflationary semantics of IQL do not allow deleting ground facts $R(v)$ from the input, even if the db-transformation we are trying to compute specifies that they are not in the output.

Following [8,7] we first consider only disjoint input-output schemas. These suffice for a general study of database *queries* and *insertions*. The disjoint input-output db-transformations (*dio-transformations*) are all db-transformations

$$\gamma \subseteq instances(S_{in}) \times instances(S_{out})$$

where: for some schema S with disjoint projections S_{in}, S_{out} and for every $(I, J) \in \gamma$ we have that I, J are projections of *one* instance of S on S_{in}, S_{out}.

For *relational* schemas, the dio-transformations (by definition) are the same as the graphs of computable queries as defined in [18]. For *acceptors* (programs that answer *yes*, *no* or *loop for ever*) we use the set of *yes/no db-transformations*: these are all db-transformations with an output schema consisting of a single relation of type the empty tuple.

Proposition 4.1 Each dio-transformation for relational schemas is the semantics of some IQL program.

Proposition 4.2 Each yes/no db-transformation is the semantics of some IQL program.

For dio-transformations, we generalize Proposition 4.2 to obtain completeness of IQL "up to copy elimination". We show that given a dio-transformation γ, there is an IQL program which on input I_0 constructs finitely many copies of images of I_0 through γ. These copies are guaranteed to be identical up to renaming of the oid's and are distinguishable from each other (but we don't know whether one of these copies can be selected by an IQL program).

One may attempt a direct analogy between TM's and IQL programs. A function is TM computable iff its graph is TM acceptable. To show this, one uses the fact that: a TM can easily enumerate the integers. The enumeration of instances in IQL is not as simple. So unfortunately (unlike TM's) from the fact that yes/no db-transformations are expressible one cannot directly derive the fact that arbitrary db-transformations can be computed by IQL programs. This is the reason for the technical restriction of up to copy elimination, which we formalize below. Note how the different copies are separated by using distinct sets of oid's, given explicitely in a new relation.

Definition 4.2 Let S be a schema with classes $\{P_1, ...P_n\}$ and I an instance of S. We define $\bar{S}$, the *schema for copies* of S by augmenting S with a single new relation name $\bar{R}$ with associated type $\{P_1 \vee ... \vee P_n\}$. An instance $\bar{I}$ of $\bar{S}$ is an *instance with copies of* I if there are O-isomorphic copies $I_1, \ldots, I_n$ of I such that:

1. $ground\text{-}facts(\bar{I}[S]) = ground\text{-}facts(I_1) \cup \ldots \cup ground\text{-}facts(I_n)$,
2. $\bar{I}(\bar{R}) = \{objects(I_1), ..., objects(I_n)\}$, where $objects(I_i)$ $(i = 1, \ldots, n)$ are pairwise disjoint.

We say that binary relation γ is binary relation $\bar{\gamma}$ *up to copy* when we have that:

(a) if $(I_0, I) \in \gamma$ then for some $\bar{I}$, $(I_0, \bar{I}) \in \bar{\gamma}$ and $\bar{I}$ is an instance with copies of I,

(b) if $(I_0, \bar{I}) \in \bar{\gamma}$ then for some I, $(I_0, I) \in \gamma$ and $\bar{I}$ is an instance with copies of I.

We now come to the principal result of this section:

Theorem 4.2 Each dio-transformation is the semantics of some IQL program up to copy.

Is copy elimination expressible in IQL? A positive answer would imply the *conjecture* that: "each dio-transformation is the semantics of some IQL program". In many important cases, this conjecture is true. Natural programs, such as the graph example of the introduction or the examples from Section 3, do not use copy elimination. More specifically, we can show the following propositions.

Proposition 4.3 If γ is a dio-transformation, such that the output schema contains no class, then γ is the semantics of some IQL program. In particular, each query in the calculus/algebra of the complex-objects model of [1] is expressible in IQL.

Proposition 4.4 If γ is a dio-transformation, such that D does not occur in the output schema, then γ is the semantics of some IQL program.

Proposition 4.5 Each query in the calculus/algebra of the Logical Data Model of [37] is the semantics of some IQL program.

As in [7], one can introduce *nondeterministic* dio-transformations and a nondeterministic variant of IQL that expresses all of them. (With nondeterminism, selection of one out of a set of copies is easy.) Also one can in the style of [7] consider "ordered databases" and prove completeness of IQL in this setting.

IQL has inflationary semantics and is a simple and elegant model for queries and insertions. However, because of monotonicity it cannot express deletions of ground facts from the input. Let IQL* be the language obtained by allowing negative facts in heads of rules and interpreting them as *deletions* in the style of the "*"-languages of [7]. IQL* allows manipulation of arbitrary input-output schemas. All the propositions, remarks and open questions above can be extended analogously, from disjoint to nondisjoint input-output schemas.

5 On the Sublanguages of IQL

A major strength of IQL is that it contains, as syntactically defined sublanguages, many popular, declarative database query formalisms. For example with small modifications of the syntax: on relations, we can identify Datalog, relational calculus and Datalog with negation (stratified or inflationary); on complex-objects, we can identify the restricted calculus of [1] and

COL with range-restriction [3]. All these sublanguages have PTIME *data-complexity*: *each fixed program can be evaluated in time that is polynomial in the input instance size.* (The size of an instance is assumed to be the size of some standard encoding of the instance.)

In this section, we use syntactic conditions to obtain the sublanguages IQL^{rr} and IQL^{pr}, where $IQL^{rr} \subset IQL^{pr} \subset IQL$. These sublanguages have PTIME data-complexity. Also, all the above popular query formalisms are contained in IQL^{pr}. The containment is proper, because many queries on cyclic schemas (such as the graph example from the introduction) are IQL^{rr} expressible. We first consider the ptime-restricted language.

Definition 5.1 A program is *ptime-restricted* if all its rules are ptime-restricted. A rule is ptime-restricted if all variables occurring in its body are ptime-restricted. Let r be a rule.

1. Each variable of a type without the set constructor is ptime-restricted in r.
2. If all variables in t_1 are ptime-restricted and $t_1(t_2)$, $t_1{=}t_2$, $t_2{=}t_1$ is a positive literal in the body of r then all variables in t_2 are also ptime-restricted.

Our further choices are determined by the features of Example 3.2: (a) variables must be restricted, and (b) invention must be controlled. First, a program is *invention-free* if no variable occurs in the head and not the body of a rule.

Let Γ be an IQL program, such that the leftmost symbol of each rule is a relation name. Γ is *recursion-free* if the directed graph $G(\Gamma)$ is acyclic. The nodes of $G(\Gamma)$ are the relation and class names of the program; there is an arc in $G(\Gamma)$ from a node n to a node n' if in some rule r: (1) n is a relation name R or class name P occurring in *body*(r), or n is a class name P and a variable of type P occurs in *body*(r), and (2) n' is the leftmost symbol R' in the rule, or n' is some class name P' occurring in $\mathbf{T}(R')$, where R' is the leftmost symbol of the rule.

Based on "invention-freedom", "recursion-freedom" and composition (;) we define:

Definition 5.2 An IQL^{pr} program Γ is an IQL program if it is of the form $\Gamma_1;...;\Gamma_k$ where for each i in $1 \ldots k$, Γ_i is ptime-restricted and is either recursion-free or invention-free.

We could have chosen more involved criteria. The rationale for our definition is that it is simple, subsumes most popular query languages and suffices to show the principal result of this section.

Theorem 5.1 Each query expressed by an IQL^{pr} program can be answered in time polynomial in the size of the input instance.

In IQL^{pr}, we have guaranteed program termination and polynomial data-complexity. From a practical standpoint this is not enough. It is also critical to limit the possible valuations of variables to o-values that already exist in the database. Because, searching over type interpretations built from the constants in the database is in theory polynomial, but in practice it is too expensive. This additional requirement leads naturally to the definition of range-restriction and to IQL^{rr} where instead of Def. 5.1(1) above, one uses: "each variable of type P for some class P is range-restricted".

6 Type Inheritance

In this section, we add type inheritance to our object-based data model. Let us extend the definition of schema as follows:

Definition 6.1 A schema S is a triple $(\mathbf{R}, \mathbf{P}, \mathbf{T}, \leq)$, where $\mathbf{R}$ is a finite set of relation names, $\mathbf{P}$ is a finite set of class names, $\mathbf{T}$ is a function from $\mathbf{R} \cup \mathbf{P}$ to *types*$(\mathbf{P})$ and $\leq$ is a partial order on $\mathbf{P}$ called *isa hierarchy.*

A common use of type inheritance is to specify, by the addition of an *isa* hierarchy to the schema, that certain classes share certain structural properties. We observe that it is possible to express this intended meaning of *isa* statements, by schemas and instances without *isa.* The main reason is the inclusion of union types in our type system. So, given *union* types, one can think of *isa* as a convenient shorthand.

We proceed in two steps. As a first step, we reexamine one of our basic assumptions namely, *the disjointness of oid assignments.*

Inherited oid assignments: To preserve static type checking, it seems necessary to know a priori what type of result we get at evaluation time, when we valuate terms such as $\hat{x}$. This means static knowledge of, which classes oid's can belong to. We formalize this static knowledge using a common engineering intuition: "oid's are created in a single class and automatically belong to the ancestors of this class in the *isa* hierarchy."

Definition 6.2 Let $\mathbf{P}$ be a set of class names and $\leq$ a partial order on $\mathbf{P}$. An *inherited oid assignment* $\bar{\pi}$ for $\mathbf{P}$ is an oid assignment, for which there exists a disjoint oid assignment π such that:
$\bar{\pi}(\mathrm{P}) = \cup \{ \pi(P') \mid P' \in \mathbf{P}, P' \leq P \}$
(for each P in $\mathbf{P}$).

At this point, we must deal with the meaning of $\mathbf{T}$ (the types in the schema) given inherited oid assignments. Let us first examine a frequently quoted example.

Example 6.1 Let our schema contain four classes $P_1 \equiv$ *person* with $\mathbf{T}(P_1) = [name{:}D]$, $P_2 \equiv$ *student* with $\mathbf{T}(P_2) = [name{:}D,\ course\text{-}taken{:}D]$, $P_3 \equiv$ *instructor*, with $\mathbf{T}(P_3) = [name{:}D,\ course\text{-}taught{:}D]$, and $P_4 \equiv$ *ta*, with $\mathbf{T}(P_4) = [name{:}D,\ course\text{-}taught{:}D,\ course\text{-}taken{:}D]$.

With a disjoint oid assignment, we can capture the meaning of *ta*'s (these are $\pi(ta)$), of *instructors* who are not *ta*'s (these are $\pi(instructor)$) etc.

But we would also like to say that: every *ta isa student* and *instructor*, every *student isa person*, and every *instructor isa person*. This is expressed by the partial order $P_4 \leq P_3$, $P_4 \leq P_2$, $P_3 \leq P_1$, $P_2 \leq P_1$ and can be realized by using the inherited oid assignment $\bar{\pi}$.

Now the type information in $\mathbf{T}$ must apply to $\bar{\pi}$ and not to π. For instance, if R is a relation of type $[A_1{:}student, A_2{:}instructor]$, we expect to find in R tuples $[o,o']$ where o is in $\pi(student)$ and o' in $\pi(instructor)$, but also such tuples with o in $\pi(ta)$ and o' in $\pi(instructor)$ etc. □

In the previous example, there is no restriction on the types of classes related via the *isa* relationship. Namely, *isa* and types are declared separately and independently. Given $\bar{\pi}$, the type interpretations (as defined in Section 2) determine what are the possible o-values. Thus, instances can be defined by a single modification of Definition 2.4: in Conditions (1) and (2) of Definition 2.4 use

$$[\![\mathbf{T}(\cdot)]\!]_{\bar{\pi}} \text{ instead of } [\![\mathbf{T}(\cdot)]\!]_{\pi}$$

where $\bar{\pi}$ is the oid assignment inherited from π.

Wherever $\bar{\pi}(P)$ appears in the modified definition it can be replaced by $\cup \{\ \pi(P') \mid P' \leq P\ \}$. This corresponds to replacing, in the types, a class P by the disjunction of its "smaller-or-equal" classes, e.g., replacing *student* by *student* $\vee$ *ta*.

***-Interpretations:** The purpose of type inheritance is to specify structure sharing. So we cannot reasonably assume that *isa* and types are declared separately and independently. Interestingly, their interaction does not significantly complicate things and can be captured using a $*$-interpretation in the style of [16]: In all cases, replace $[\![\ldots]\!]_{\pi}$ by $[\![\ldots]\!]_{\pi *}$ except for tuples where,

$$[\![[A_1 : \tau_1, \ldots, A_k : \tau_k]]\!]_{\pi *} = \{[A_1 : v_1, \ldots, A_k : v_k, A_{k+1} : v_{k+1}, \ldots, A_l : v_l] \mid \text{for some } A_{k+1}, \ldots, A_l, (l \geq k) \text{ distinct from } A_1, \ldots, A_k \text{ and } v_i \in [\![\tau_i]\!]_{\pi *},\ i = 1, \ldots, k\}.$$

Observe that, in this definition, $v_{k+1}, \ldots, v_l$ are o-values of totally unconstrained types. One can show an analogue to Proposition 2.1:

Proposition 6.1 For each type expression (1) there is an intersection reduced, $*$-equivalent type expression, (2) there is an intersection free, $*$-equivalent over disjoint oid assignments type expression.

Let us come back to the canonical example.

Example 6.2 A more succinct specification of the schema of Example 6.1 is as follows,
person has-type $\tau_1 = [name{:}D]$,
student has-type $\tau_2 = [course\text{-}taken{:}D]$
and *student isa person*,
instructor has-type $\tau_3 = [course\text{-}taught{:}D]$
and *instructor isa person*
ta isa student and *ta isa instructor*.
The intention here is to have the *isa* hierarchy force a certain structural similarity. For this interpret the types using $*$-interpretations of types given $\bar{\pi}$, where $\bar{\pi}$ is the inherited oid assignment. The type of *person* is τ_1, the type of *student* is $\tau_1 \wedge \tau_2$, the type of *instructor* is $\tau_1 \wedge \tau_3$, and the type of *ta* is $\tau_1 \wedge \tau_2 \wedge \tau_3$. Using Proposition 6.1, we can eliminate the intersection and get the type expressions explicitly given in Example 6.1.

Clearly, the $*$-interpretation forces some compatibility of the types of classes connected via *isa*. Otherwise, their conjunction may end up being the empty type or some trivial type. □

Following through with this kind of approach, one can find a serious drawback with exclusive use of $*$-interpretations. It leads to legal instances with attributes that do not appear in the schema. Thus, there is insufficient information in the schema to describe the instance and, consequently, little hope of finding a complete query language according to our requirements.

This suggests a blend of the two possibilities. One would like to use the starred interpretation to force inheritance of structure. But, one would also like to use the unstarred interpretation on the disjoint oid assignment π that generates $\bar{\pi}$. For instance, the value of an object in $\pi(ta)$ in Example 6.1 should have exactly type $[name{:}D,\ course\text{-}taught{:}D,\ course\text{-}taken{:}D]$, and no other attributes.

In this spirit, let P be in $\mathbf{P}$. We construct τ_P such that:

$$[\![\tau_P]\!]_{\bar{\pi}*} = \cap \{\ [\![\mathbf{T}(P')]\!]_{\bar{\pi}*} \mid P \leq P'\ \}.$$

Such a type expression τ_P exists by Proposition 6.1. Now we can put everything together in a new definition for instance, that gives meaning to *isa*'s.

Definition 6.3 An instance I of schema $(\mathbf{R}, \mathbf{P}, \mathbf{T}, \leq)$ is a triple (ρ, π, ν), where ρ is an o-value assignment for $\mathbf{R}$, π is a disjoint oid assignment for $\mathbf{P}$, and ν is a partial function from the set $\cup \{\ \pi(P) \mid P \in \mathbf{P}\ \}$ to o-values, such that:

1. $\rho(R) \subseteq [\![\mathbf{T}(R)]\!]_{\bar{\pi}}$ (for each $R \in \mathbf{R}$),
2. $\nu(\pi(P)) \subseteq [\![\tau_P]\!]_{\bar{\pi}}$ (for each P in $\mathbf{P}$).
3. if $\mathbf{T}(P) = \{\tau\}$ then ν is total on $\pi(P)$ (for each $P \in \mathbf{P}$).

where $\bar{\pi}$ is the oid assignment inherited from π.

The language IQL can now be used with no modification.

7 A Value-Based Data Model

In this section we introduce a value-based data model and relate it to the object-based model of the previous sections. We use a simplified framework: only class names $\mathbf{P}$ and *v-types*($\mathbf{P}$). The set *v-types*($\mathbf{P}$) consists of all type expressions in *types*($\mathbf{P}$) constructed without $\vee, \wedge, \emptyset$ i.e., we assume only base, finite set and tuple construction.

The value-based schemas have the form ($\mathbf{P}$,$\mathbf{T}$) and should be compared to object-based schemas of the form ($\emptyset$,$\mathbf{P}$,$\mathbf{T}$). For simplicity both are denoted ($\mathbf{P}$,$\mathbf{T}$).

We also limit consideration to IQL programs from input schema S to output schema S', where S, S' are disjoint value-based schemas. We use IQL^v for this subset of IQL.

The *pure values* that are considered here can be defined as trees, in the style of o-value representations. They have the same kinds of nodes (base, finite set, finite tuple) but there are two differences: (1) no oid's occur in them, (2) they might have infinite depth. These *infinite trees* are variants of the infinite trees in [21]. For example, the set nodes do not have a fixed arity and the order of their children is not significant, whereas all functions in [21] do have a fixed arity. However, using the fact that the sets that are considered are finite, it is an easy but tedious exercise to show that: the properties of the infinite trees in [21] also hold for pure value infinite trees.

The assignments and type interpretations of Section 2 have analogs in the value-based case. Given a set of class names $\mathbf{P}$, a *finite assignment* I for $\mathbf{P}$ is a function from $\mathbf{P}$ to *finite* sets of pure values. Each finite assignment I defines a function from *v-types*($\mathbf{P}$) to sets of pure values, called the type *interpretation* given I.

This function $[\![\cdot]\!]_I$ is analogous to $[\![\cdot]\!]_\pi$ of Section 2 (i.e., $[\![\mathrm{P}]\!]_I = I(P)$, for each P, and $[\![\cdot]\!]_I$ extends to *v-types*($\mathbf{P}$) by structural induction.)

Definition 7.1 A *v-schema* S is a pair ($\mathbf{P}$,$\mathbf{T}$), where $\mathbf{P}$ is a finite set of class names and $\mathbf{T}$ is a function from $\mathbf{P}$ to *v-types*($\mathbf{P}$) such that:
(*) for each $P \in \mathbf{P}$, $\mathbf{T}(P)$ is not a class name.
A *v-instance* I over ($\mathbf{P}$,$\mathbf{T}$) is a finite assignment for $\mathbf{P}$ such that: $I(P) \subseteq [\![\mathbf{T}(P)]\!]_I$ for each $P \in \mathbf{P}$.

Note the simplicity of these definitions, that generalize complex-object data models by adding cyclicity to both schemas and instances. The technical condition (*) on the $\mathbf{T}$ of a v-schema is imposed to avoid pathological cases, such as $\mathbf{T}(P_1) = P_2$ which does not specify any structure for P_1.

A regular tree is a tree with a finite number of subtrees [21]. An important consequence of the finiteness of assignments is that each value occurring in a v-instance is a regular tree.

Proposition 7.1 Each pure value occurring in a v-instance is a regular tree. □

Now let us compare object-based and value-based instances over schema ($\mathbf{P}$,$\mathbf{T}$). It is simple to define translations of pure values into objects and vice-versa: φ can be thought of as producing o-values by adding oid's and ψ as producing pure values by loosing oid's. In the absence of union types, we can show that these translations preserve information in the following sense:

Proposition 7.2 For each v-instance I, $\psi\varphi I = I$. □

Propositions 7.1 and 7.2 have some interesting consequences about computable queries in the value-based model. Regularity guarantees the existence of a simple encoding of v-instances on Turing-machine tapes, so it is possible to compute. Genericity is defined in the usual way. A *vdio-transformation* is a transformation from v-instances over a v-schema S to v-instances over a disjoint v-schema S', which is recursively enumerable and generic. A language is *vdio-complete* if it expresses exactly the vdio-transformations.

Let Γ be an IQL^v program from v-schema S to disjoint v-schema S'. Γ transforms $\varphi(I)$ into some instance J. We shall also say that: Γ transforms I into $\psi(J)$. So, by using Γ for the value-based model we mean that it is preceded by the fixed transformation φ and followed by the fixed transformation ψ. First, note that this defines a mapping. It is easy to check that this mapping is recursively enumerable and generic. Recall that IQL can express all dio-transformations (up to copy). Using this completeness theorem, and noticing that automatic copy elimination is performed in ψ we have:

Theorem 7.1 IQL^v is vdio-complete. □

Cyclicity in schemas and instances has been treated using one basic idea: "class names are part of the type syntax". Is it possible to separate class names and class types? This removal of the P's from type expressions would give us a notion of the "pure" structure of a class (which might be useful for determining coercion strategies).

A natural semantic definition for the *pure type* of a class P in a schema S would be $\{v \mid \exists$ instance I over S such that $v \in I(P)\}$, i.e., the set of all values that

may be members of P. In [6] we exhibit a constructive definition of a pure type and show that the two definitions are equivalent. Let us present the basic intuition via an example.

Example 7.1 Consider the following example:
$\mathbf{T}(P_1) = [A_1{:}D, A_2{:}D]$,
$\mathbf{T}(P_2) = [A_3{:}D, A_2{:}P_1]$, and
$\mathbf{T}(P_3) = [A_1{:}D, A_2{:}P_3]$.
Intuitively, the pure type of P_1 is given by $[A_1{:}D, A_2{:}D]$, and of P_2 by $[A_3{:}D, A_2{:}[A_1{:}D, A_2{:}D]]$.

On the other hand, the pure type of P_3 should be some recursive type with certain regularities. If one were to use recursive syntax here, a fixpoint type constructor would have to be added, e.g., $\mu x.\tau$, and its fixpoint semantics specified. An important point is that: the straightforward least fixpoint constructions do not work. In this example, such constructions would give empty sets for P_3. The problem is that such approaches specify finite trees, and we want v-instances with infinite trees. Perhaps greatest fixpoints would be more appropriate.

Instead of introducing recursive syntax it is simpler to define the pure type of P_3 using a sequence of finite types which are better and better approximations of P_3 [21]:

$$[A_1{:}D, A_2{:}\Omega], \quad [A_1{:}D, A_2{:}[A_1{:}D, A_2{:}\Omega]],$$
$$[A_1{:}D, A_2{:}[A_1{:}D, A_2{:}[A_1{:}D, A_2{:}\Omega]]], \ldots$$

The basic intuition is to consider these types as sets of partially defined finite trees (they have leaves labeled $\perp$). Now the constructive definition of pure type of P_3 is: "the set of regular trees that are limits of sequences of partially defined trees". □

Acknowledgements: we want to thank the people in the Altaïr and Verso groups for fruitful discussions and arguments, and in particular, François Bancilhon, Claude Delobel, Sophie Gamerman, Stéphane Grumbach, Christophe Lécluse, Philippe Richard and Fernando Velez. We also thank Maria-Teresa Otoya for pointing-out that object relations have been studied for a century in psychology.

References

[1] S. Abiteboul and C. Beeri. On the Manipulation of Complex Objects. INRIA Technical Report, 1987.

[2] S. Abiteboul and S. Grumbach. COL: a Logic-based Language for Complex Objects. In *Proc. EDBT*, 271–293, 1988.

[3] S. Abiteboul, S. Grumbach, A. Voisard, and E. Waller. An Extensible Rule-based Language with Complex Objects and Data-functions. In *Proc. DBPL-II Workshop*, Oregon, 1989. To appear.

[4] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM TODS*, 12:525–565, 1987.

[5] S. Abiteboul and R. Hull. Data-functions, Datalog and Negation. In *Proc. ACM SIGMOD*, 143–153, 1988.

[6] S. Abiteboul and P. Kanellakis. *Object Identity as a Query Language Primitive* INRIA Techical Report, March 1989.

[7] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Language. In *Proc. ACM PODS*, 240–250, 1988.

[8] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Update and Specification. In *Proc. ACM PODS*, 260–268, 1987.

[9] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[10] F. Bancilhon. Object-Oriented Database Systems. In *Proc. ACM PODS*, 152–162, 1988.

[11] F. Bancilhon et al. The Design and Implementation of O_2, an Object-Oriented Database System. In *Proc. OODBS2 Workshop*, Badmunster RFA, 1988.

[12] F. Bancilhon, S. Cluet, and C. Delobel. Query Languages for Object-Oriented Database Systems. In *Proc. DBPL-II Workshop*, Oregon, 1989. To appear.

[13] F. Bancilhon and S. Khoshafian. A Calculus for Complex Objects. In *Proc. ACM PODS*, 53–60, 1986.

[14] J. Banerjee et al. Data Model Issues for Object-Oriented Applications. *ACM TOIS*, 5:1:3–26, 1987.

[15] C. Beeri and al. Sets and Negation in a Logic Database Language (LDL1). In *Proc. ACM PODS*, 21–37, 1987.

[16] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[17] M.J. Carey, D.J. Dewitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD*, 413–423, 1988.

[18] A. Chandra and D. Harel. Computable Queries for Relational Data Bases. *JCSS*, 21:2:156–178, 1980.

[19] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.

[20] T. Codd. Extending the Database Relational Model to Capture more Meaning. *ACM TODS*, 4:4:397–434, 1979.

[21] B. Courcelle. Fundamental Properties of Infinite Trees. *TCS*, 25, 95–169, 1983.

[22] E. Dahlaus and J. Makowski. Computable Directory Queries. In *Proc. CAAP*, 1986. LNCS 214, Springer-Verlag.

[23] D. Fishman et al. Iris: an Object-Oriented Database Management System. *ACM TOIS*, 5:1:46–69, 1987.

[24] P. Fischer and S. Thomas. Operators for Non-first-normal-form Relations. In *Proc. COMPSAC*, Chicago USA,1983.

[25] A. Goldberg and D. Robson. *Smalltalk 80, the Language and Implementation.* Addison-Wesley, 1983.

[26] R. Hull. Relative Information Capacity of Simple Relational Schemata. *Siam J. of Computing*, 15:3, 1986.

[27] R. Hull and J. Su. Untyped Sets, Invention and Computable Queries. In *Proc. ACM PODS* 1989. To appear.

[28] B. Jaeschke and H.J. Schek. Remarks on the Algebra of Non-first-normal-form Relations. In *Proc. ACM PODS*, 124–138, 1982.

[29] P. Kanellakis. *Elements of Relational Database Theory.* Brown U. Technical Report, 1988. To appear as a chapter in the Handbook of Theoretical Computer Science.

[30] S. Khoshafian and G. Copeland. Object Identity. In *Proc. OOPSALA*, 1986.

[31] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proc. ACM PODS*, 1989. To appear.

[32] W. Kim. *A Foundation for Object Oriented Databases.* Technical Report, MCC, 1988.

[33] P.G. Kolaitis and C.H. Papadimitriou. Why not Negation by Fixpoint? In *Proc. ACM PODS*, 231–239, 1987.

[34] H.F. Korth, M.A. Roth, and A. Silberschatz. *Extended Algebra and Calculus for not 1NF Relational Databases.* U. Texas Austin Technical Report, 1985.

[35] G.M. Kuper. Logic Programming with Sets. In *Proc. ACM PODS*, 11-20, 1987.

[36] G.M. Kuper. *The Logical Data Model: a New Approach to Database Logic.* Stanford U., PhD thesis, 1985.

[37] G.M. Kuper and M.Y. Vardi. A New Approach to Database Logic. In *Proc. ACM PODS*, 86–96, 1984.

[38] C. Lecluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. ACM PODS*, 1989. To appear.

[39] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proc. ACM SIGMOD*, 424-434, 1988.

[40] D. Maier. A Logic for Objects. In *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, Washington USA, 1986.

[41] D. Maier, A. Otis, and A. Purdy. Development of an Object-Oriented Dbms. *Quarterly Bulletin of IEEE on Database Engineering*, 8, 1985.

[42] H. Schek and M. Scholl. The Relational Model with Relation-valued Attributes. *Information Systems*, 1986.

[43] J.D. Ullman. Database Theory - Past and Future. In *Proc. ACM PODS*, 1–10, 1987.

[44] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[45] J. Verso. *Verso: a Database Machine Based on non-1NF Relations.* To appear in Nested Relations and Complex Objects *Springer-Verlag.*

[46] S. Zdonik. Object Management Systems for Design Environments. *Quarterly Bulletin of IEEE on Database Engineering*, 8, 1985.