Object Identity as a Query Language Primitive

SERGE ABITEBOUL

I.N.R.I.A.-Rocquencourt, Chesney Cedex, France

AND

PARIS C. KANELLAKIS

Brown University

Abstract. We demonstrate the power of object identities (oids) 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 oids 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 type checked, can be evaluated bottom-up, and naturally generalizes most popular rule-based 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.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classification—*object-oriented languages*; H.2.1 [**Database Management**]: Logical Design; H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems—*object-oriented databases*

General Terms: Languages, Theory

Additional Key Words and Phrases: Computable query, inheritance, object identity, object-oriented database, query language, regular tree, rule-base language

---

A preliminary version of this paper appears in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data* (Portland, Ore.). ACM, New York, pp. 159–173.

Paris C. Kanellakis died tragically in an airplane crash on December 20, 1995. A technical obituary can be found in *ACM Computing Surveys*, March 1996.

The work of S. Abiteboul was supported by the Projet de Recherche Coordonnée BD3.

Authors' address: S. Abiteboul, INRIA/Rocquencourt, Domaine de Voluceau, Rocquencourt-BP 105, 78153 Chesnay Cedex France, e-mail: Serge.Abiteboul@inria.fr.

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.

The notion of object and object-oriented features have seriously modified the database field over the past ten years. This happened in a variety of ways. First, object extensions have been introduced in the main relational systems, for example, in Oracle. Also, in a clear departure from the relational model, object-oriented database systems [Bancilhon 1988] that integrate the relational and object technology have been commercialized, for example, O_2 or ObjectStore. Somewhere in between, some constructors proposed the so-called object-relational systems that provide richer modeling facilities than traditional relational systems.

The integration of objects in databases in these various contexts has been the focus of a great deal of experimentation and research since the late eighties.[1] These developments in databases are largely based on concepts and software tools from object-oriented programming [Goldberg and Robson 1983; Bancilhon 1988; Kim 1988]. More generally, the integration of programming languages and database systems is an important research activity.

Unfortunately, the understanding of the "principles of object-oriented databases" is still rather limited. This is in marked contrast with the relational model [Codd 1979] that is now equipped with an elegant and relevant theory [Abiteboul et al. 1995; Ullman 1988; Kanellakis 1998].

This paper together with papers such as Beeri [1990], Denninghoff and Vianu [1991], or Breazu-Tannen et al. [1992] participate in clarifying some of the foundations of object-oriented databases. Our contribution is to demonstrate that the concept of *object identity* (*oid*) is a powerful programming primitive for database query languages by: having *oids as the centerpiece of a data model with a rich type system, inheritance and a powerful query language*, called *Identity Query Language* (*IQL*).

A major motivation for our work was the study of the formal aspects of the O_2 system [Bancilhon et al. 1988]. The standard data model for object-oriented data model as defined by the *Object Data Management Group* [Cattell 1994] was strongly influenced by the O_2 system. As a consequence, our model bears a lot of resemblance with the ODMG data model. The ODMG model does not have union types. We believe that union type is essential and we use it here. On the other hand, we consider here a single kind of collections, set collections, whereas the ODMG also handles list, bags, and arrays. The extension of the theory presented here to these other kinds of collections is nontrivial.

Oids have been part of many data models; for example, they are called *surrogates* in Codd [1979], *l-values* in Kuper and Vardi [1993], or *object identifiers* in Abiteboul and Hull [1987]. They have been highlighted as an essential part of object-oriented database systems [Khoshafian and Copeland 1986]. A variety of reasons have been given for their use, for example, structure sharing, updates

[1] See, for example, Atkinson and Buneman [1987], Maier et al. [1985], Zdonik [1985], Banerjee et al. [1987], Fishman et al. [1987], Carey et al. [1988], and Bancilhon et al. [1988].

[Abiteboul and Hull 1987], or the encoding of cyclicity [Kuper and Vardi 1993]. We use oids 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, oids are "typed pointers" and IQL is based on a *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 [Codd 1970], most complex-object data models[2], and the logical data model (LDM) [Kuper and Vardi 1993; Kuper 1985]. It can be viewed as the common upper bound of the models used in Kuper and Vardi [1993] and Abiteboul and Beeri [1988]. 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 rule based, (2) it can be 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 Chandra and Harel [1980] 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 Kuper and Vardi [1993], Abiteboul and Beeri [1988], and Abiteboul and Grumbach [1988] and corresponds to strong typing in programming languages. A number of recent language proposals in this area do not have these properties. In many cases, there is no instance-schema separation, and in some the query languages can be viewed as untyped extensions of Prolog.[3]

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*; that is, values containing oids, and (2) a *partial function* ν of oids to o-values, this mapping is the essence of the data model. Oids and constants are o-values, but so are finite trees built out of constants and oids 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 oids yield *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 oids, where these oids are mapped

[2] See, for example, Abiteboul and Beeri [1988], Thomas and Fischer [1986], Jaeschke and Schek [1982], Korth et al. [1985], Schek and Scholl [1986], and Verso [1986].

[3] See, for example, Bancilhon and Khoshafian [1986], Maier [1986], Kifer and Wu [1989], and Kifer and Lausen [1989].

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 oids.

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, for example, Datalog. Relations can name subsets of classes and function as useful temporaries. Also, this distinction allows a direct generalization of both Kuper and Vardi [1993] and Abiteboul and Beeri [1988].

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

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

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

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

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

Note that the types refer to base domain, for *string*, and to class names, *1st-generation*, but not to relation names. Also, note the cyclicity in the type associated with *1st-generation* 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 oids. Classes are assigned disjoint sets of oids. The partial function ν assigns o-values to the oids of the instance. Each one of these oids 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 each $\nu(o)$ for o in $\pi(P)$ is in $\mathbf{T}(P)$.

In instance I, we denote the oids as *adam*, *eve*, *cain*, *abel*, *seth*, *other*. (The object *other* denotes some other child of *adam* and *eve* of whom nothing is known. This can indeed be viewed as a place holder for possibly several such persons.) Note that *adam* is distinct from string Adam. I is cyclic; see this by following the ν mapping of the oids.

π(*1st-generation*) = {*adam*, *eve*},

π(*2nd-generation*) = {*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*}],

$\nu(eve)$ = [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 Abiteboul and Grumbach [1988], for the manipulation of sets, and the detDL language of Abiteboul and Vianu [1988], for the invention of new oids. 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 [Abiteboul and Vianu 1988; Kolaitis and Papadimitriou 1988] combined with set/tuple types, invention of new oids, and a weak form of assignment. Inflationary semantics has been chosen because of its simplicity and its generality as a control flow 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. For example, a directed graph may be represented as a binary relation whose tuples are the arcs of the graph or as a class whose type is recursive. In the second representation, each node has an oid, a name, and a set of descendant nodes. IQL allows converting the first representation into the second and vice-versa. Thus, it is possible *to go from acyclic to cyclic schemas*. The following example illustrates most of the features of IQL on this very transformation.

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 a class 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 oids 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:

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

$$R_0(x) \leftarrow R(y, x).$$

In the second stage, we produce two oids per node using a semantics in the style of detDL [Abiteboul and Vianu 1988]. We use a relation R' with $\mathbf{T}(R') =$

$[A_1: D, A_2: P, A_3: P']$ whose tuples contain oids from class P and from another class P'. Class P' is a class with $\mathbf{T}(P') = \{P\}$; that is, it's oids have values that are sets of oids from P. The following rule invents two oids for each node, one of which will go into class P and the other into class P'

$$R'(x, p, p') \leftarrow R_o(x).$$

Note how the variables p, p' in the head are not in the body. When the new oids 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').

Note that this is essentially equivalent to using Skolem functors in oids denotation as introduced by Maier [1986], and refined for instance in Hull and Yoshikawa [1991], Kifer and Lausen [1989], and Kifer et al. [1993]. In a Skolem based approach, one would instead use a rule such as:

$$R'(x, f(x), f'(x)) \leftarrow R_o(x),$$

where $f(x)$, $f'(x)$ would denote the two new oids. In practice, the use of Skolem functors sometimes facilitate programming. From a theoretical viewpoint, they can be easily simulated and their use would unnecessarily force us to migrate from atomic oids to more complex (constructed) oids.

In the third stage, we nest the oids representing nodes in P into sets of successors of a node. This nesting of elements q is done by using the oids p' of P' as temporary names. Each p' is set valued and its value, noted $\hat{p}'$, is a set in which the corresponding qs are collected. This dereferencing and assignment to objects in P' simulates the effect of a COL data-function [Abiteboul and Grumbach 1988] or a grouping in LDL [Beeri et al. 1987].

$$\hat{p'}(q) \leftarrow R'(x, p, p'), R'(y, q, q'), R(x, y).$$

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-oids as a second component. This weak form of assignment is performed only when $\hat{p}$ was undefined (see Abiteboul and Hull [1988]). No further changes are made to $\hat{p}$.

$$\hat{p} = [x, \hat{p'}] \leftarrow R'(x, p, p').$$

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 needs to be modified by separating it with a (;) from the rest of the rules. □

An important primitive in the language is the invention of oids. This serves a triple goal: (1) objects may be part of the result and oids must be assigned to them, (2) invented oids are used for set manipulation, (3) they are also used to obtain completeness in the sense of Chandra and Harel [1980]. 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 [Beeri et al. 1987], *data-functions* in COL [Abiteboul and Grumbach 1988], or *universal quantification* [Kuper 1987]. Thus, one of our contributions is to show that *the manipulation and creation of sets can be realized only using invented oids*.

We examine (3) in detail in Section 4. The notion of completeness of [Chandra and Harel 1980] is adapted to our context. Intuitively, the language must capture all transformations that are recursively enumerable and that preserve some isomorphism properties [Chandra and Harel 1980; Hull 1986]. Completeness results have been shown for the relational [Chandra and Harel 1980; Abiteboul and Vianu 1987/1998; 1988/1998] and for many complex-object data models [Dahlaus and Makowski 1986; Abiteboul et al. 1987; Hull and Su 1989].

Our notion of completeness is more general than the notion used in Chandra and Harel [1980] and Abiteboul and Vianu [1988/1998]. However, on relational schemas, the two notions coincide. The originality of our extension comes from the presence of oids: two instances are viewed as identical if they are isomorphic up to renaming of oids. 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 we show that this technical restriction is necessary. We have to add a choice construct to reach completeness for disjoint input-output schemas. To obtain completeness for nondisjoint schemas, we also need to add noninflationary features to IQL. These are based on the study of deletions in Abiteboul and Vianu [1988/1998].

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, that is, with PTIME *data-complexity*. In Abiteboul et al. [1989], similar restrictions are used in the context of the COL language to obtain queries that are evaluated in PTIME.

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 is 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 typability of IQL programs. However, the removal of this condition is necessary if one is to study *type inheritance* as proposed in Cardelli [1988]. 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 use of the union type constructor is 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. Oids 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 [Courcelle 1983]. Our analysis allows us to show that IQL can serve as a language for this model as well. Object identities, in this context, lose all semantic denotation to become purely primitives of the language. This is a nontrivial link between value-based and object-based [Ullman 1988]. A value-based point of view can be used to understand *pure-values* (no oids) and *equality-by-value* (as a coercion mechanism for realizing inheritance).

There are aspects of object-oriented database systems that our mathematical model cannot capture. For example, O_2 emphasizes programming in a modular fashion, by having *methods* attached to classes and by accessing data only through these methods (*encapsulation*). (See Abiteboul et al. [1995].) Moreover, sharing of programs is possible via *method inheritance*. We view these issues as largely orthogonal to the one of: "what should be the computational capabilities of a query language for an object-oriented database?" In a last section, Section 8, we briefly consider how our study clarifies this latter issue.

2. *An Object-Based Data Model*

2.1. PRELIMINARIES. 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 oids* $O = \{o_1, o_2, \ldots\}$. Throughout our exposition, we use the generic notation $[A_1: \ldots, \ldots, A_k: \ldots]$ (where k is a nonnegative 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 2.1.1. 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: $[A_1: v_1, \ldots, A_k: v_k]$ and $\{v_1, \ldots, v_k\}$.

Definition 2.1.2. Let **R** be a finite set of relation names and let **P** be a finite set of class names.

An *o-value assignment* for **R** is a function ρ mapping each name in **R** to a *finite* set of o-values.

An *oid assignment* for **P** is a function π mapping each name in **P** to a *finite* set of oids, and we call π *disjoint* if $P \neq P'$ implies $\pi(P) \cap \pi(P') = \emptyset$ (where P, $P' \in \mathbf{P}$).

Since o-values are defined using finite tupling and finite setting, it is possible to represent them using *finite trees* of a special form. These trees have three kinds of nodes: (1) nodes labeled by an element of $D \cup O$ and having no children, (2) nodes labeled by a $\times$ and having a finite number $k \geq 0$ of children, where the arcs to these children are labeled by distinct attributes, (3) nodes labeled by a $\star$ and having a finite number $k \geq 0$ of children, where the arcs to these children are unlabeled. We also require that the children of a $\star$-labeled node be roots of distinct subtrees; this guarantees the elimination of duplicates from our representation of sets.

By finiteness, each *relation* $\rho(R)$ $(R \in \mathbf{R})$ and each *class* $\pi(P)$ $(P \in \mathbf{P})$ is itself an o-value, representable by a finite tree with a $\star$-labeled root. Note that both our o-value and oid assignments give names to *sets* of o-values. The relations here resemble those of the relational data model. But classes are used in a fundamentally different way in our subsequent definitions of types, database schemas, and instances.

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

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

Given a type expression and an oid assignment, the *type* (in a more traditional programming language style) can be obtained. 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 $[\![(\tau_1 \wedge \tau_2)]\!]_\pi = [\![\tau_1]\!]_\pi \cap [\![\tau_2]\!]_\pi$,
—$[\![\{\tau\}]\!]_\pi = \{\{v_1, \ldots, v_j\} \mid j \geq 0, \text{ and } v_i \in [\![\tau]\!]_\pi, i = 1, \ldots, j\}$,
—$[\![[A_1\colon \tau_1, \ldots, A_k\colon \tau_k]]\!]_\pi = \{[A_1\colon v_1, \ldots, A_k\colon v_k] \mid v_i \in [\![\tau_i]\!]_\pi, i = 1, \ldots, k\}$.

We may 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.

Proposition 2.2.1. For each type expression, (1) there is an intersection reduced, equivalent type expression and (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. The proof of the proposition is by straightforward algebraic manipulation of parse trees and by using the semantics of type expressions. We omit the actual tedious argument and instead illustrate it

by some examples:

$$[A_1\colon D, A_2\colon \{P_1\}] \wedge [A_1\colon D, A_2\colon \{P_2\}] \text{ and } [A_1\colon D, A_2\colon \{(P_1 \wedge P_2)\}]$$

are equivalent over all π, and they are equivalent over all disjoint π to $[A_1\colon D, A_2\colon \{\emptyset\}]$. Also $(\{D\} \vee P_1) \wedge P_2$ is equivalent over all π to $(P_1 \wedge P_2)$ and over all disjoint π to $\emptyset$. Note the difference between the type expressions $\{\emptyset\}$ and $\emptyset$. $[A_1\colon \emptyset]$ and $\emptyset$ are equivalent, but $\{\emptyset\}$ and $\emptyset$ are not.

2.3. Database Schemas and Instances. In this section, we present schemas and instances and comment on their definitions.

Definition 2.3.1. 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 $\mathbf{R} \cup \mathbf{P}$ to *type-exp*(**P**).

A schema can be alternatively specified using a syntax of the form:

$$\textit{relation} \quad R_1\colon \{\tau_1\}, \ldots R_m\colon \{\tau_m\}$$

$$\textit{class} \quad P_1\colon \{\tau'_1\}, \ldots P_n\colon \{\tau'_n\}$$

In our notation, $\mathbf{R} = \{R_1, \ldots, R_m\}$, $\mathbf{P} = \{P_1, \ldots, P_n\}$, and for each i, $\mathbf{T}(R_i) = \tau_i$ and $\mathbf{T}(P_i) = \tau'_i$.

Definition 2.3.2. 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 $\cup \{\pi(P) \mid P \in \mathbf{P}\}$ of oids to o-values, such that:

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

Continuing with $R_1\colon \{\tau_1\}, \ldots R_m\colon \{\tau_m\}$ and $P_1\colon \{\tau'_1\}, \ldots P_n\colon \{\tau'_n\}$, let us assume that the value of R_i, that is, $\rho(R_i)$, is $\{t_{i1}, \ldots\}$ and that of P_i, that is, $\nu(P_i)$ is $\{o_{i1}, \ldots\}$. Then type constraints are that each t_{ij} is of type τ_i and each $\nu(o_{ij})$ of type τ'_i.

More precisely, the partial function ν associates o-values to the oids in the instance (i.e., the oids occurring in the range of π). When ν is defined for o, $\nu(o)$ denotes the *value* of o; $\hat{o}$ is a useful alternative notation for $\nu(o)$. From Conditions (1) and (2) of Definition 2.3.2, it follows that the sets named in the schema "contain" (for relations) and "contain pointers to" (for classes) o-values of the appropriate type. Condition (3) is a technical one that will be justified below. For an illustration of the definitions, see the *Genesis* example in the introduction.

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 Definition 2.3.2 and from the semantics of types.

Let $I = (\rho, \pi, \nu)$ be an instance of a schema $S = (\mathbf{R, P, T})$. 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. By Condition (3) in Definition 2.3.2, ν must be total for set valued oids. 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 nonset 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 oids occurring in I; *constants*(I) for the set of all constants occurring in I. By Definitions 2.1.1 and 2.1.2, the sets *ground-facts*(I), *constants*(I), *objects*(I) are finite.

The structural part of the model generalizes that of many previously introduced data models:

—The *relational data model* is the special case of our model with types only of the restricted form $[A_1: D, \ldots, A_k: D]$, $k \geq 0$, and schemas of the form (**R**, ∅, **T**).

—Our model generalizes most *complex-object*[4] *data models*, where their schemas are of the form (**R**, ∅, **T**) with types more general than for the relational case.

—The *logical data model* (LDM) is also a special case of our model. It corresponds to schemas of the form (∅, **P**, **T**) where the types are trees of bounded depth. (For instance, the value of an object cannot be a tuple where one component is itself a tuple.) In Appendix B of Kuper [1985], an attempt is made to formulate LDM in a fashion closer to our model. A problem with that approach is the requirement of having only classes and forcing some of them to behave like relations, through semantic restrictions. The resulting restrictions for duplicate elimination are quite involved. Directly expressing duplicate elimination is one motivation for the dichotomy of **R** and **P**.

Finally, we have already commented on the relationship of this work with the O_2 data model and system. We conclude with two remarks.

Remark 2.3.3. Incomplete information can be modeled using oids with undefined value. So our model has some capability for expressing incomplete data, even without complex machinery as in, e.g., Bancilhon and Khoshafian [1986]. Besides this, there is an important technical reason for having oids with undefined values. The language IQL builds objects in stages, and oids with undefined values are needed in the intermediate stages. So, during the construction of the value of an object, that value will often be incomplete even if

[4] Unfortunately *object* is an overloaded word. The objects of object-oriented programming are qualitatively different and less rigorously defined from what is commonly referred to in the database literature as complex-objects. We reserve the term *complex-object* for the later case.

eventually it does become complete. When we know nothing about a set-valued object, we assume here that it is the empty set. This is the rationale for the simplifying assumption we make in Condition (3) in the instance definition. Of course, as standard with incomplete information, one could choose a more refined model that would allow to distinguish between knowing nothing about a set and knowing that it is empty. Since incomplete information is not the focus of the present paper, we will ignore this here.

Remark 2.3.4. From the point of view of semantic data modeling, we provide types and some functional constraints through the ν mapping. In Section 6, we add inheritance. More applications can be modeled by adding dependencies to the schema (e.g., relational functional dependencies or statements like *for each x the spouse of the spouse of x is x*). A first-order-logic in the style of Kuper and Vardi [1993] and Abiteboul and Beeri [1988] and others can be used. Indeed, the language IQL can form the basis of such a logic.

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 *type-exp*($\mathbf{P}$). The *terms* and their types are ($k \geq 0$):

—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 $t_1, \ldots, t_k$ terms of type τ, $\{t_1, \ldots, t_k\}$ is a term of type $\{\tau\}$,
—for $t_1, \ldots, t_k$ terms of types $\tau_1, \ldots, \tau_k$, $[A_1\colon t_1, \ldots, A_k\colon t_k]$ is a term of type $[A_1\colon \tau_1, \ldots, A_k\colon \tau_k]$.

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 $L \leftarrow L_1, \ldots, L_k$ $(k \geq 0)$, where L is a literal called $head(r)$ and $L_1, \ldots, L_k$ is a sequence of literals called $body(r)$ and:

(1) $head(r)$ is a fact and thus is typed,
(2) each literal in $body(r)$ is either typed; or is of the form $t_1 = t_2$ 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 3.1.1. 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 allow coercion with respect to 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 simplify the presentation as in Chandra and Harel [1980]. Constants can be added easily without changing the framework (see Abiteboul and Vianu [1987]).

3.2. SEMANTICS. The semantics of program $\Gamma(S, S_{in}, S_{out})$ is a binary relation $\gamma(\Gamma)$ on instances. The pair (I, I') is in $\gamma(\Gamma)$ 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* $\gamma_\infty(\Gamma)$.

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 Abiteboul and Vianu [1988]. 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 nonset valued oids 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.

—$\theta R = \{v \mid R(v) \in ground\text{-}facts(I)\}$ and $\theta P = \{o \mid P(o) \in ground\text{-}facts(I)\}$,

—$\theta\hat{x} = \{v \mid \widehat{\theta x}(v) \in ground\text{-}facts(I)\}$ where $\hat{x}$ is of set type,

—$\theta\hat{x} = v$ if $\widehat{\theta x} = v$ is in $ground\text{-}facts(I)$ where $\hat{x}$ is not of set type,

—$\theta\{t_1, \ldots, t_k\} = \{\theta t_1, \ldots, \theta t_k\}$, $\theta[A_1: t_1, \ldots, A_k: t_k] = [A_1: \theta t_1, \ldots, A_k: \theta t_k]$ $(k \geq 0)$.

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 $\theta t_2 \notin \theta t_1$, (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:

$$val\text{-}dom(\Gamma, I) = \{(r, \theta) \mid r \in \Gamma, I \models \theta body(r),$$

$$\theta \text{ is a valuation exactly on variables in } body(r),$$

$$\text{and there is } no \text{ extension } \bar{\theta} \text{ of } \theta \text{ such that } I \models \bar{\theta}\, head(r)\}$$

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 oids and constants, or inventing new oids. Here are the laws governing the invention of oids:

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 oids for each parallel branch).

Inflationary Operators. Given a program Γ, the *inflationary one-step operator* $\gamma_1(\Gamma)$ is a binary relation on instances. The pair of instances (I, J) is in $\gamma_1(\Gamma)$ if there exists a valuation-map η for Γ and I and:

$$ground\text{-}facts(J) = ground\text{-}facts(I) \cup \{\eta(r, \theta)head(r) \mid$$

$$\text{for some } r, \theta \text{ subject to } (*) \text{ below}\} \cup \{P(o) \mid$$

$$\text{for some } r, \theta \text{ and } x \text{ of type } P, o = \eta(r, \theta)x \text{ and } o \text{ is invented}\}$$

where

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

The statements in ($\star$) are motivated as follows: We want to achieve a deterministic, inflationary semantics. So, we cannot modify a value for an object o when one has already been determined. Furthermore, we cannot choose between several alternative values when they are derived at the same step of the computation.

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

Programs are determinate. Because of the quantification over valuation-maps η in the definition of $\gamma_1(\Gamma)$, the binary relation $\gamma_1(\Gamma)$ contains (I, J) pairs for all possible legal choices of invented oids. (Note that, if the valuation-domain is empty, there is only one trivial valuation-map and $J = I$). It follows that, there could be many J associated to a single instance I in $\gamma_1(\Gamma)$, $\gamma_\infty(\Gamma)$, and $\gamma(\Gamma)$. However, as we shall see in Theorem 4.1.3, all these J are isomorphic to each other. Also, by the definition of $\gamma_\infty(\Gamma)$, there may be no finite sequence leading to a fixpoint. Therefore, IQL programs are *determinate*; they define partial functions up to renaming of oids. For a more formal treatment, see Section 4.

Naive inflationary evaluation. 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 oids. The semantics in terms of binary relations can be thought of as: all the possible outputs that one would get by naive inflationary evaluation of the rules on the input.

3.3. TYPE CHECKING. The syntax and the semantics of IQL impose a number of typing restrictions on programs. Typing restrictions, verifiable through *type checking*, may be introduced in a database language for a variety of reasons. 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*, for example, 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."

IQL programs can be type-checked. In order to justify our claim, consider the naive inflationary evaluator of the rules. Before the evaluation is started, the type of each variable is known. Thus, there is an upper bound on the values each variable can be instantiated to. The only side-effects of the program involve the derivation of new ground facts. These side-effects will produce legal instances because of the well-typedness condition on heads of rules. This condition and all other typing restrictions can be easily checked.

There is one exception. Namely, the treatment of ground facts $\hat{o} = v$ 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, deciding at this inexpensive check is needed in some evaluation is not recursive; see Abiteboul and Hull [1988].

All terms of the IQL language are typed. Having to declare the type information for each term would make the programs tedious to write and would hide the simplicity of the rules. As will be made clear from the examples in the next subsection, most of the type information can be omitted. Automatic *partial type inference*, based on a number of shorthand conventions, can replace explicit declarations.

3.4. SHORTHANDS AND EXAMPLES. We accept $R(t_1, \ldots, t_k)$ as an alternative notation for $R([A_1: t_1, \ldots, A_k: t_k])$, 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* [Abiteboul and Vianu 1988]. The differences between detDL and IQL restricted to relations are: slightly different semantics for valuation-domains and invented constants in detDL versus invented oids in IQL. However, it is very simple to simulate detDL in IQL.

It is shown in Abiteboul and Vianu [1988] 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* [Abiteboul and Grumbach 1988; Abiteboul et al. 1989]. The next example shows the IQL realization. For better clarity, we use capital letters, for example, X, Y, for set variables.

Example 3.4.1. Let **(R, P, T)** be a schema, $R_1, R_2, R_3 \in \mathbf{R}$,

$$\mathbf{T}(R_1) = \mathbf{T}(R_3) = [A_1: D, A_2: \{D\}], \text{ and } \mathbf{T}(R_2) = [A_1: D, A_2: D].$$

We want to unnest R_1 into R_2, and then nest R_2 into R_3. For unnesting, we use the single rule:

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

For nesting, we 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:

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

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

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

and Γ_2:

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

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 *parameterized 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 oids. 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 Thomas and Fischer [1986], Jaeschke and Schek [1982], Schek and Scholl [1986], and Abiteboul and Beeri [1988] and the calculus queries of Abiteboul and Beeri [1988] and Korth et al. [1985] 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 Abiteboul and Beeri [1988] 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 3.4.2. 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:

$$R_1(X) \leftarrow X = X,$$

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 oids. 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 $[A_1\colon \{D\}, A_2\colon \{D\}, A_3\colon P]$.

The powerset program consists of the rules:

$$R_1(\{\ \}) \leftarrow$$

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

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

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

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

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

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 oids occurs in a "loop." Such recursion with invention of oids may clearly be the cause of nonterminating computations. For instance, let R_3 be a relation with $\mathbf{T}(R_3) = [A_1\colon P, A_2\colon P]$. Then, the rule

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

may cause the nontermination 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 oids bounded by a polynomial in the size of the input, composition, and weak assignment to non-set oids.

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 3.4.3. Consider the two schemas:

S has only one class P with $\mathbf{T}(P) = P \vee [A_1\colon P, A_2\colon P]$ and

S' has only one class P' with $\mathbf{T}'(P') = [B_1\colon \{P'\}, B_2\colon \{[A_1\colon P', A_2\colon P']\}]$.

We use one temporary relation R with $\mathbf{T}(R) = [C_1\colon P, C_2\colon P']$ and omit the attributes, when there is no ambiguity.

The first program we describe encodes an instance with union types into an instance without union types. Applying the second program on the output of the first produces an instance identical (up to renaming of the oids) with the input of the first program. Thus, no information is lost when using the first program.

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

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

$$\hat{x}' = [\{y'\}, \emptyset] \leftarrow R(x, x'), R(y, y'), y = \hat{x}$$

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

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

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

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

$$\hat{x} = w \leftarrow R(x, x'), R(y, y'), R(z, z'), [y, z] = w, \hat{x}' = [\emptyset, \{[y', z']\}].$$

Note the use of coercions in the bodies. For instance, in the first program, $\hat{x}$ 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. *On the Expressive Power of IQL*

Only some binary relations on instances can be transformations defined by database programs. For example, instances must be well typed. To formalize what the meaningful binary relations are, we extend the notion of database (db-) transformation of Abiteboul and Vianu [1988] and, thus, the notion of computable relational query of Chandra and Harel [1980]. The only departure from the classical notion is that functionality is weakened by allowing transformations that are determinate up to renaming of oids.

In this framework, we investigate the expressive power of IQL. We show that each IQL program expresses a db-transformation. For *disjoint* input–output schemas, we show that all db-transformations are expressible, *up to copy elimination*, in IQL. We show that copy elimination cannot be expressed in general. This surprisingly demonstrates that the concept of determinate transformations introduced here affects in a nontrivial manner the standard notion of query computability.

Finally, we show how completeness can be obtained using a "deterministic-choice" primitive and consider the case of nondisjoint input–output schemas.

4.1. DB-TRANSFORMATIONS.

In this section, we extend the standard notion of database computability.

Consider a one-to-one mapping h from $D \cup O$ onto $D \cup O$ that maps D to D and O to O. Such a mapping can be extended to o-values and instances in the obvious manner. It transforms an instance into an *isomorphic* instance. We call such a mapping, a *DO-isomorphism*. A bijection h over O can be viewed as a DO-isomorphism by extending it with: $hd = d$ for each d in D. We call such bijections *O-isomorphisms*. Clearly, an O-isomorphism can also be viewed as an isomorphism over o-values and instances. Similarly, one can consider D-isomorphism.

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 oids, see Chandra and Harel [1980] and Hull [1986]. The fourth condition is new and expresses a form of *functionality*.

Definition 4.1.1. 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) for each DO-isomorphism h, $(I, J) \in \gamma$ implies that $(hI, hJ) \in \gamma$, and
(4) $(I, J_1), (I, J_2) \in \gamma$ imply that there exists an O-isomorphism h' such that $h'J_1 = J_2$.

Let us now comment on Condition (4) above. Observe that it can be replaced with the seemingly more general condition: "$(I_1, J_1), (I_2, J_2) \in \gamma$ and I_1, I_2 are O-isomorphic imply that J_1, J_2 are O-isomorphic." One can show that the resulting definition is equivalent with the one used here.

It follows from Conditions (3)–(4) that no new constants can appear in the output. More precisely,

$$\text{if } (I, J) \text{ is in a db-transformation } \gamma, \text{ then } \mathit{constants}(J) \subseteq \mathit{constants}(I).$$

In contrast, the kind of functionality enforced by Condition (4) allows the presence of oids in the output that were not in the input. This is a significant addition to the frameworks of Abiteboul and Vianu [1988/1998] and Chandra and Harel [1980]. It is important to be able to create new oids 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 oids as atomic elements are irrelevant, only their interrelationships matter. Consider the IQL example of the introduction. The oids 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: IQL programs define only db-transformations. It follows that IQL programs are determinate in the sense of Condition (4). We first illustrate by an example why, if one is to guarantee soundness, the disjointness of oid assignments is important.

Example 4.1.2. Consider a schema with two classes P_1, P_2 with $\mathbf{T}(P_1) = \{D\}$ and $\mathbf{T}(P_2) = \{\{D\}\}$. For example, an object in P_1 has value a set of strings and an object in P_2 has value a collection of sets of strings. Suppose nondisjoint oid assignments were allowed in legal instances. It is possible to have an oid o belonging to both $\pi(P_1)$ and $\pi(P_2)$, and $\nu(o) = \{\ \}$ in a legal instance I. Now consider an IQL program with a rule $\hat{x}(z) \leftarrow \cdots$, and a rule $\hat{y}(\{z, z'\}) \leftarrow \cdots$, where x has type P_1, y has type P_2, and z, z' have type D. Clearly such a program has well-typed heads of rules but, if x and y are both instantiated to o, it produces an illegal instance. The main problem here is that for nondisjoint oid assignment, we cannot be sure of the type of the terms $\hat{x}$ and $\hat{y}$, when they are instantiated during evaluation.

One could argue that the problem highlighted in this example is due to the polymorphism of the empty set. But it is possible to create more involved examples. Also, similar problems would arise if instead of one "base" type D, many, nondisjoint ones were allowed.

The use of the disjointness simplifies the task of type checking. Note however that a weaker restriction could have been used. For instance, one could have

allowed the same oid to be assigned to two distinct classes P_1, P_2 if $\mathbf{T}(P_1) = \mathbf{T}(P_2)$. □

THEOREM 4.1.3. *The semantics of an IQL program is a db-transformation.*

PROOF. Let γ be the semantics of an IQL program Γ. We have to show that γ is a db-transformation. First, Condition (2) of Definition 4.1.1 is obviously satisfied.

Condition (1) is satisfied because of the typing of the heads of rules. A fine point is that this satisfaction does depend on the fact that the oid assignment is disjoint. The disjointness guarantees that each oid belongs to a unique class and that, when it is assigned a value, the value is of the correct type.

Consider Condition (3). Let (I, J) be in γ, h be a DO-isomorphism, and $I_0 = I, \ldots, I_n = I', J = I'[S]$ be a derivation of J on input I. Each derivation step corresponds to one application of the one-step operator γ_1, except for the last one which is a projection. It is easy to see that $hI = hI_0, \ldots, hI_n = hI', hJ = hI'[S]$ is a derivation of hJ on input hI. Therefore Condition (3) is satisfied.

Finally, let (I, J_1) and (I, J_2) be in γ. A straightforward induction on derivation length suffices to show that the derivations of J_1 and J_2 are isomorphic. Thus, Condition (4) is also satisfied. □

This soundness theorem raises a natural completeness question: are all db-transformations expressible in IQL? Consider a relation name R belonging to both the input and output schemas. A problem is that the inflationary semantics of IQL does not allow the deletion of ground facts from R, even if the db-transformation that we are trying to compute specifies that they are not in the output. A Turing Machine (TM) analog for the inflationary semantics could be a *nonerasing* TM. Following Abiteboul and Vianu [1987; 1988], we first consider only disjoint input-output schemas. This simplifies the task since we can ignore the problem of deletion: we introduce a fact in the output only when we are sure that this fact belongs to the answer.

4.2. DISJOINT INPUT-OUTPUT SCHEMAS: QUERIES AND INSERTIONS. In this section, we focus on nonerasing transformations and disjoint input–output schema. We show that IQL is complete "up to copy elimination."

The disjoint input–output db-transformations (*dio-transformations*) are all db-transformations

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

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}. Note that this implies that the set of oids in the input and output are disjoint. So, for example, having as output the input itself is not a dio-transformation, but having as output an O-isomorphic copy of the input is a dio-transformation.

For *relational* schemas, the dio-transformations (by definition) are the same as the graphs of computable queries as defined in Chandra and Harel [1980]. 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. Two propositions about IQL easily follow from the literature, when IQL programs are limited either to

(1) query programs for the relational data model or to (2) acceptors for arbitrary inputs.

PROPOSITION 4.2.1. *Each dio-transformation for relational schemas is the semantics of some IQL program.*

PROOF. The proof follows the proof of Abiteboul and Vianu [1988] for showing that each relational transformation can be expressed in the language detDL. The differences between the languages detDL and IQL restricted to relations do not affect the essence of the proof. □

PROPOSITION 4.2.2. *Each yes/no db-transformation is the semantics of some IQL program.*

PROOF. The instance is first encoded by an IQL program in a relational schema. Oids are invented to denote more structured o-values. The encoding performed is an obvious representation of ground facts and is easy to realize in IQL. Then, Proposition 4.2.1 is used to conclude. □

For dio-transformations, we use Proposition 4.2.2 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 oids and are distinguishable from each other.

In the next definition, note how the different copies are separated by using distinct sets of oids, given explicitly in a new relation.

Definition 4.2.3. Let S be a schema with classes $\{P_1, \ldots, P_n\}$ 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 $\{P_1 \vee \cdots \vee P_n\}$. 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) $\mathit{ground\text{-}facts}(\bar{I}[S]) = \mathit{ground\text{-}facts}(I_1) \cup \cdots \cup \mathit{ground\text{-}facts}(I_n)$,
(2) $\bar{I}(\bar{R}) = \{\mathit{objects}(I_1), \ldots, \mathit{objects}(I_n)\}$, where $\mathit{objects}(I_i)$ $(i = 1, \ldots, n)$ are pairwise disjoint.

We say that a binary relation γ is the 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 4.2.4. *Each dio-transformation is the semantics of some IQL program up to copy.*

To prove this theorem, we use two lemmas:

LEMMA 4.2.5. *Each dio-transformation, whose output schema has a single class and no union types, is the semantics of some IQL program up to copy.*

PROOF. Let γ be a dio-transformation, $(I_0, I) \in \gamma$, and let $S = (\{R_1, \ldots, R_n\}, \{P\}, \mathbf{T})$ be the output schema.

Let P' be a new class and $\mathbf{T}'$ be such that for each i in $1 \ldots n$, $\mathbf{T}'(R_i)$ is obtained from $\mathbf{T}(R_i)$ by substituting P by P' everywhere. An instance (ρ, π, ν) over S can be represented (up to oid renaming) by a tuple

$$[A_0\colon \pi'(P'), A_1\colon \rho'(R_1), \ldots, A_n\colon \rho'(R_n),$$
$$A\colon \{[A_1\colon o, A_2\colon \nu'(o)] \mid o \in P', \nu'(o) \text{ defined}\}]$$

of type

$$\tau = [A_0\colon \{P'\}, A_1\colon \{\mathbf{T}'(R_1)\}, \ldots, A_n\colon \{\mathbf{T}'(R_n)\}, A\colon \{[A_1\colon P', A_2\colon \mathbf{T}(P')]\}].$$

where (ρ', π', ν') is O-isomorphic to (ρ, π, ν). A finite set of instances over S can be represented (up to oid renaming) by a relation whose elements have type τ.

By Proposition 4.2.2, there is a program $\Gamma_{y/n}$ which given an instance I_0 and a tuple of type τ representing an instance I, checks whether $(I_0, I) \in \gamma$.

The computation of $\bar{\gamma}$ (such that γ is $\bar{\gamma}$ up to copy) is realized by an IQL program $\bar{\Gamma}$ as follows. The program $\bar{\Gamma}$ consists of three parts and on input I_0 proceeds as follows.

Consider the total ordering of pairs of positive integers (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) $\cdots$ The first part of $\bar{\Gamma}$ visits each pair of integers in the above order; this exhaustive enumeration can be done as in Chandra and Harel [1980] and Abiteboul and Vianu [1987] by realizing counters over a unary alphabet. For a pair (i, j), this part first invents i oids. Then, it constructs a relation of elements of type τ representing the set $\mathcal{T}_i$ of all instances over S, that can be constructed using the i oids and constants from the input. This "encoding" construction is possible, because the finite sets of o-values to be constructed are just the interpretations of types restricted to the constants from the input. This encoding is easily seen to be realizable in IQL, using terms for finite tupling and finite subsetting. (Note that the union of types requires a different treatment.)

Let us come to the second part of $\bar{\Gamma}$. The idea of the ordered visiting of integer pairs above is that the first component represents the number i of oids in the output instance, and the second one, the number j of steps taken by program $\Gamma_{y/n}$ to accept. So in this part $\bar{\Gamma}$ uses $\Gamma_{y/n}$ to compute the set $\mathcal{T}_{i,j}$ of tuples in $\mathcal{T}_i$ representing instances that are images of I_0 by γ, validated by $\Gamma_{y/n}$ in j steps. Two cases occur:

(1) $\mathcal{T}_{i,j}$ is empty. Then Γ increments (i, j) and does another iteration of the first part.
(2) $\mathcal{T}_{i,j}$ is not empty. Then relation $\mathcal{T}_{i,j}$ contains the tuple representations of some of the images of I_0 by γ and the third part is used.

By the fact that γ is a dio-transformation, one has that the tuples in $\mathcal{T}_{i,j}$ are all O-isomorphic representations of I. The third part of $\bar{\Gamma}$ transforms these tuples into $\bar{I}$ an instance with copies of I. The oid invention and the weak assignment of IQL suffice for this "decoding."

Now the two conditions for γ to be $\bar{\gamma}$ up to copy are satisfied: the first condition because of the exhaustive enumeration of instances, and the second because $\Gamma_{y/n}$ provides the only halting condition. □

LEMMA 4.2.6. *Each dio-transformation, whose output schema has a single class, is the semantics of some IQL program up to copy.*

PROOF. In the definition of union types, we have used a binary $\vee$ constructor. Assume here that the parse trees of types have been put in a canonical form, where $\vee$-nodes have arbitrary arity, but only non-$\vee$ nodes as children. This does not change the semantics because of the associativity and commutativity of $\cup$.

Consider the schema S' obtained from S by replacing the class P by a class P' everywhere in S and by replacing (inductively) each union of types $\tau = \tau_1 \vee \cdots \vee \tau_m$ by $[A_1\colon \{\tau_1\}, \ldots, A_m\colon \{\tau_m\}]$. Our intention is to represent an o-value v of type τ by tuple $[A_1\colon \alpha_1, \ldots, A_m\colon \alpha_m]$, where for each i in $1 \cdots n$, α_i is $\{v\}$ when v is of type τ_i and is $\emptyset$ otherwise. Let *rep* be the one-to-one function, which maps an instance over S to an instance over S' that represents S in the above fashion. Let γ be a dio-transformation, $(I_0, I) \in \gamma$ and also let,

$$\gamma' = \{(I_0, rep(I)) \mid (I_0, I) \in \gamma\}.$$

Clearly, γ' is a dio-transformation without the union of types and without multiple classes in the output schema. By Lemma 4.2.5, there is an IQL program Γ_1 computing it up to copy. Consider the transformation γ'' defined as follows:

$$\gamma'' = \{(rep(I), I') \mid I' \text{ an } O\text{-isomorphic copy of } I \text{ with } objects(I') \cap objects(I) = \emptyset\}.$$

It is easy to see that γ'' is a dio-transformation and that there is a simple IQL program Γ_2 computing this "decoding" transformation. For an example of such a decoding program, see Example 3.4.3. Note that here, in treating union, we use untyped equality literals in bodies.

The program $\Gamma_1;\Gamma_2$ computes γ up to copy. We use the fact that composition can be simulated in IQL, so ";" can be viewed as a meta construct of the language. □

PROOF OF THEOREM 4.2.4. Let γ be in a dio-transformation with output schema $S = (\mathbf{R}, \mathbf{P}, \mathbf{T})$ and where $\mathbf{P} = \{P_1, \ldots, P_n\}$. Let $R_1, \ldots, R_n$ be new relation names, for each R in $\mathbf{R}$ let $\hat{R}$ be a new relation name, and let P be a new class name. Consider the schema $S' = (\mathbf{R}', \{P\}, \mathbf{T}')$ where:

(1) $\mathbf{R}' = \{R_1, \ldots, R_n\} \cup \{\hat{R} \mid R \in \mathbf{R}\}$
(2) $\mathbf{T}'(R_i) = P$, for each i in $1 \ldots n$,
(3) $\mathbf{T}'(\hat{R})$, for each $R \in \mathbf{R}$, is obtained from $\mathbf{T}(R)$ by replacing each $P_1, \ldots, P_n$ by P,
(4) $\mathbf{T}'(P) = \tau_1 \vee \cdots \vee \tau_n$ where for each i in $1 \cdots n$, τ_i is obtained from $\mathbf{T}(P_i)$ by replacing each $P_1, \ldots, P_n$ by P.

An instance $I = (\rho, \pi, \nu)$ over S can be coded by an instance $rep'(I) = (\rho', \pi', \nu)$ over S' as follows:

(1) $\pi'(P) = \pi(P_1) \cup \cdots \cup \pi(P_n)$,

(2) for each R in $\mathbf{R}$, $\rho'(\hat{R}) = \rho(R)$,
(3) for each i in $1 \cdots n$, $\rho'(R_i) = \pi(P_i)$.

Assume that $(I_0, I) \in \gamma$. Like in Lemma 4.2.6, the dio-transformation γ can be decomposed into two transformations:

$$\gamma' = \{(I_0, rep'(I)) \mid (I_0, I) \in \gamma\}.$$

$\gamma'' = \{(rep'(I), I') \mid I'$ an O-isomorphic copy of I with $objects(I')$

$$\cap\ objects(I) = \emptyset\}.$$

To conclude, it suffices to notice that γ' can be computed up to copy by an IQL program by Lemma 4.2.6, and the decoding involved by γ'' can also be realized in IQL. Again here there is a need for untyped equality literals in rule bodies. □

We will show in Section 4.3 that copy elimination cannot be expressed in IQL, and present solutions to that limitation in Section 4.4. However, the previous result allows us to conclude that for many interesting cases, IQL has a quite sufficient power. Natural programs, such as the graph example of the introduction or the examples from Section 3, do not require copy elimination. More specifically, we can show the following propositions.

PROPOSITION 4.2.7. *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 Abiteboul and Beeri* [1988] *is expressible in IQL.*

PROOF. The important observation is that, because there are no oids in the output, the copy elimination is automatic. More specifically, in the proof of Lemma 4.2.5, there is no P and all tuple representations are identical. □

PROPOSITION 4.2.8. *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.*

PROOF. By inspection and modification of the proof of Lemma 4.2.5. Since the output instance contains only oids, we can enumerate instances instead of sets of instances like in Lemma 4.2.5. To do that, in step (i, j), the i oids that are considered are invented in some precise order and this order is remembered. Using this order, the tuples of type τ representing instances of S with i oids can be enumerated. Thus, there is no need for copy elimination. □

PROPOSITION 4.2.9. *Each query in the calculus/algebra of the Logical Data Model of Kuper and Vardi* [1993] *is the semantics of some IQL program.*

PROOF. In LDM, there is limited invention of oids, but output schemas are constrained. It is simple to simulate all the algebraic operators of LDM in IQL directly, that is, without any exhaustive enumeration of instances or copies. Thus, copy elimination is not necessary for simulating LDM. □

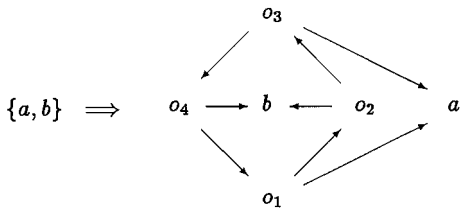

FIG. 1. A query not expressible in IQL.

4.3. COPY ELIMINATION

Did the hen make the egg, or was it the egg that made the hen?

In this section, we show that copy elimination cannot be realized in IQL. The intuition is as follows:

The input is a set of two elements a, b. The output is a directed quadrangle of new objects with a connected to the two objects of a diagonal and b to those of the other one. Consider two adjacent vertices of the quadrangle, o_1 (the hen) and o_4 (the egg). Intuitively, which one should have been created first? The conclusion is that they must have been created at the same time. Then how can we choose a direction for the arrow between them without violating genericity?

THEOREM 4.3.1. *There is a dio-transformation whose output has a single class that is not the semantics of any IQL program.*

PROOF. Consider the two schemas: S has a single relation R of type D; S' consists of a single class P' of type $\emptyset$, and a single relation R' of type $[B: P', C: D \vee P']$.

The dio-transformation γ is defined by: for each (I, I') in γ, $I = (\rho, \emptyset, \emptyset)$, $I' = (\rho', \pi', \nu')$, and the following hold:

(α) if $\#(\rho(R)) \neq 2$, $\rho'(R') = \pi'(P') = \emptyset$;

(β) otherwise, let $\rho(R) = \{a, b\}$. Then (see Figure 1), $\pi'(P') = \{o_1, o_2, o_3, o_4\}$ for some o_1, o_3, o_2, o_4 and $\rho'(R')$ is the relation:

$$\{\langle o_1, a\rangle, \langle o_3, a\rangle, \langle o_2, b\rangle, \langle o_4, b\rangle, \langle o_4, o_1\rangle, \langle o_3, o_4\rangle, \langle o_2, o_3\rangle, \langle o_1, o_2\rangle\}$$

We first show:

CLAIM 4.3.2. *γ is a dio-transformation.*

PROOF OF CLAIM 4.3.2 (SKETCH). Clearly, γ satisfies (1, 2, 4) of the definition of db-transformation. Let (I, I') be in γ and h a DO-isomorphism. Clearly, in case (α), $(hI, hI') \in \gamma$, so (3) holds. To see that (3) also holds in case (β), it suffices to note that there is an automorphism h_0 on I': $h_0(a) = b$, $h_0(b) = a$, $h_0(o_1) = o_4$, $h_0(o_3) = o_2$, $h_0(o_4) = o_3$, $h_0(o_2) = o_1$. □

Suppose that some IQL program P computes γ. We focus on case (β) to derive a contradiction. Let I, I' be as in (β) above. We use a "naming scheme" for the created objects. Consider a computation of P on input I. We associate with each created object a term called "v-term" as follows. Each rule has an associated symbol. We define the function ξ from created objects to v-terms as follows.

Consider, for instance, the ground instance of a rule r creating a new object o:

$$H(o) \leftarrow R_1(t_1, t_2, t_3), R_2(t_3, t_4).$$

Then the v-term of o is $\xi(o) = r(z_1, z_2, z_3, z_4)$ where each z_i is obtained by replacing in t_i each object o' by its corresponding v-term $\xi(o')$.

Remark 4.3.3. A v-term is a tree with internal nodes labelled by: set or tuple constructors, or by rule symbols. The order of the children of a set vertex is considered here as irrelevant. The leaves are labelled by a or b.

CLAIM 4.3.4. *If o, o' are two distinct objects created in the computation of P on input I, $\xi(o) \neq \xi(o')$.*

PROOF OF CLAIM 4.3.4 (SKETCH). By definition of the semantics of IQL, the same instantiation of variables occurring in the body of a rule cannot be used twice. □

Returning to the proof of the theorem, consider an enumeration $T_1, \ldots, T_k, \ldots$ of the v-terms using two constants $\{a_1, a_2\}$ when it is chosen that $a_1 < a_2$. For each i, let $val(o_i)$ be the smallest integer such that $T_{val(o_i)}$ is isomorphic to $\xi(o_i)$. To conclude the proof, it suffices to show:

CLAIM 4.3.5. For each i, j, $val(o_i) = val(o_j)$.

For suppose that Claim 4.3.5 holds. Let $val = val(o_1) = \cdots = val(o_4)$. For each i, let h_i be a bijection from $\{a_1, a_2\}$ to $\{a, b\}$ with $h_i(T_{val}) = \xi(o_i)$. Since there are only two such bijections, it is easy to see that the set $\{\xi(o_1, \ldots, \xi(o_4)\}$ has at most 2 elements. Thus, by Claim 4.3.4, the set $\{o_1, \ldots, o_4\}$ has at most two elements, a contradiction. Hence, γ is not realizable in IQL. By Claim 4.3.2, γ is a dio-transformation, which concludes the proof of the theorem.

We next show:

PROOF OF CLAIM 4.3.5 (SKETCH). The proof is by contradiction. Suppose that Claim 4.3.5 does not hold and consider the following query: for $o'_1, \ldots, o'_4$ the four objects in the answer,

(1) If $val(o'_1) < \cdots < val(o'_4)$, select the constant ($a$ or b) connected to o'_1.
(2) If $val(o'_1) = val(o'_2)$ and $val(o'_3) < val(o'_4)$, select the constant connected to o'_3.
(3) If $val(o'_1) = val(o'_2) < val(o'_3) = val(o'_4)$,
   (a) if o'_1 and o'_2 are connected to the same constant, select that constant, otherwise,
   (b) select the constant connected to the source of the edge between o'_1 and o'_2 (there is such an edge since o'_1 is connected to a constant and o'_2 to the other).
(4) If $val(o'_1) = val(o'_2) = val(o'_3) \neq val(o'_4)$, select the constant connected to o'_4.

One can write an IQL program that computes the query. The program computes an internal representation of the $val(o_i)$ for each $i \in [1 \cdot\cdot 4]$, enumerates $T_1, \ldots, T_k, \ldots$, and selects one of the constants. However, this query is not a db-transformation (it is not generic), which is a contradiction by Theorem 4.1.3. □

This concludes the proof of the theorem. □

We conclude this section with an analogy between TMs and IQL programs.

Remark 4.3.6. A function is TM computable if and only if 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 TMs) from the fact that yes/no db-transformations are expressible one cannot derive the fact that arbitrary db-transformations can be computed by IQL programs. Note that a simplistic enumeration of all the instances using constants from the input would violate genericity, and so is clearly not realizable. But one could imagine more sophisticate (generic) enumerations. Consider for instance a program that would do the following: On some input I, enumerate sets of instances $X_1, X_2, \ldots$ such that:

(1) the only constants occurring in those sets are constants from I;
(2) each instance with constants from I occurs in the sequence; and
(3) in each set X_i no two instances are O-isomorphic.

Assuming such a program exists, one would be able to use the fact that yes/no db-transformations are expressible, to show that IQL is complete. By Theorem 4.3.1, such sophisticate enumeration cannot be realized in IQL.

Remark 4.3.7. Since we observed the incompleteness of IQL in 1989, the issue of completeness of languages with object creation has generated a lot of research.[5] In particular, an elegant characterization of the determinate queries expressible in IQL was shown in Van den Bussche [1992]. The characterization is as follows (see Van den Bussche [1992] for details). For an instance K, let $Aut(K)$ denote the set of DO-automorphisms of K. For a pair of instances K, K', $Aut(\langle K, K' \rangle)$ denotes the bijections on $D \cup O$ that are automorphisms of both K and K'. A determinate query q is expressible in IQL iff for each input-output pair $\langle I, J \rangle$ in q there exists a mapping h from $Aut(I)$ to $Aut(\langle I, J \rangle)$ such that for each $\tau, \mu \in Aut(I)$,

(i) τ and $h(\tau)$ coincide on I;
(ii) $h(\tau \circ \mu) = h(\tau) \circ h(\mu)$; and,
(iii) $h(\mathbf{id}_I) = \mathbf{id}_{\langle I,J \rangle}$.

where **id** denotes the identity mapping.

The "only if" part of the theorem is in the style of the technique developed in the proof of Theorem 4.3.1. The "if" part is considerably more complex, and is based on a group-theoretic argument.

4.4. Achieving Completeness. As shown above, there exist dio-transformations that are not expressible in IQL. There are various ways of "completing" IQL:

(1) As in Abiteboul and Vianu [1988/1998], one can introduce *nondeterministic* dio-transformations and a nondeterministic variant of IQL that expresses all

[5] See, for example, Andries and Paredaens [1992], van den Bussche and van Gucht [1992], van den Bussche et al. [1992], van den Bussche and Paredaens [1991], and Denninghoff and Vianu [1991; 1993].

of them: with nondeterminism, selection of one out of a set of copies is easy. (See Remark N-IQL below.)

(2) Copy elimination is possible if an *ordering* of the constants of the input is explicitly provided. (This is important from a practical point of view, because a lexicographic order is usually provided.) For the precise definition of ordered database, one can use that of Abiteboul and Vianu [1988/1998]. Intuitively, the order on the finite set of constants of the input together with the order of creation of oids allows the enumeration of instances.

However, these two solutions present inelegances. The first one yields transformations that are nondeterministic and are therefore not dio-transformations as defined above. The second one assumes that the input constants are ordered and so, takes us also out of the original framework. However, from Theorem 4.2.4, little is needed to "complete" IQL. To conclude this section, we present IQL^+ that yields exactly the desired class of transformations. The primitive that we introduce is related to the (nondeterministic) *witness* operator of Abiteboul and Vianu [1989] and the *choice* of Naqvi and Krishnamurthy [1988]. However, it is essentially different in that it is not causing any nondeterminism.

The syntax of IQL^+ is that of IQL augmented as follows: The new symbol *choose* is a literal that may occur in a rule body. We have to reconsider the definition of valuation-map. Variables that occur in the head and not in the body of a rule are interpreted differently depending on the presence of *choose* or not in the body: let x be a variable in $head(r)$ and not in $body(r)$,

(1) if *choose* is absent from the body, the interpretation is as before,

(2) if *choose* is present in the body, $\eta(r, \theta)x$ is one of the existing objects that has the same type as x, assuming that this choice does not violate the genericity condition. (This is not complicated but possibly expensive to check.)

Note that "one of the . . ." would potentially introduce nondeterminism. However, we guarantee that the inflationary one-step operator remains generic, so that the entire computation does not violate the genericity condition. Now we have:

THEOREM 4.4.1. *The semantics of an IQL^+ program is a db-transformation. Each dio-transformation can be expressed by an IQL^+ program.*

PROOF (SKETCH). To see the first sentence, consider the transformation γ corresponding to an IQL^+ program. Property (1,3) of db-transformations are clearly satisfied. Note that it is easy to decide whether a choice made at one step introduces nondeterminism. Thus, γ is clearly r.e., so (2) holds. Finally, (†) guarantees (4).

Consider the second sentence. Let γ be a dio-transformation. An IQL^+ programs computing γ proceeds as follows:

(1) Several copies of the result are first computed in some temporary relations and classes (see Theorem 4.2.4).

(2) Each of these copies is marked by an object in a new class P_1.

(3) One object ω in P_1 is selected (unique use of *choose*).

(4) The copy marked by ω is copied to the output relations and classes. □

We conclude this section by a remark on "nondeterministic IQL."

REMARK N-IQL. Let N-IQL be the language obtained by not enforcing the determinism of the *choose*; that is, a computation is indeed allowed to choose one object out of a set even if that violates genericity. The notion of nondeterministic transformations of Abiteboul and Vianu [1987/1998] can be extended to the data structure considered here, and one can show that N-IQL is "nondeterministic complete." One can also obtain a nondeterministic complete version of IQL along the lines of the "N-" languages of Abiteboul and Vianu [1988/1998] by firing one instantiation of a rule at a time instead of firing all rules that are applicable in parallel as done in this paper.

4.5. ARBITRARY INPUT–OUTPUT SCHEMAS: DELETIONS. 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 Abiteboul and Vianu [1988/1998]. IQL* allows the manipulation of arbitrary input–output schemas.

One can show that: (1) the semantics of an IQL* program is a db-transformation and (2) each db-transformation is the semantics of some IQL* program up to copy. All the results of the previous subsections (and in particular, Theorem 4.3.1) can be extended analogously, from disjoint to arbitrary input-output schemas.

We omit the formal treatment, since the additional techniques are well known (see Abiteboul and Vianu [1987/1998; 1988/1998]).

We should note that some of the simplicity of the IQL semantics has to be lost in IQL*. Deleting an oid forces deletion of other objects that have this oid in their o-value. This of course leads to more complex, but still tractable, definitions (see the treatment of update propagation in Abiteboul and Hull [1987]). From a practical standpoint, it requires more involved evaluation mechanisms, for example, with reference counts or garbage collection.

5. *On the Sublanguages of IQL*

Queries can be written directly in IQL, they can be type checked, optimized via standard techniques and evaluated bottom-up. IQL combines such high level features, together with the power and simplicity of the general mathematical model of computation presented in the previous section. Let us concentrate here on its use as a high-level query language.

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 Abiteboul and Beeri [1988] and COL with range-restriction [Abiteboul et al. 1989]. 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 I is assumed to be the size of some standard encoding of *ground-facts*(I).)

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}.

In IQL^{pr}, we guarantee program termination and polynomial data-complexity. However, from a practical standpoint, this is not enough since, for instance, searching over set-free 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}. We next define ptime-restricted and range-restricted; their difference is in Condition (1) of their respective definitions.

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 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.

For instance, Example 3.4.1 is ptime-restricted.

Definition 5.2. A program is *range-restricted* if all its rules are range-restricted. A rule is range-restricted if all variables occurring in its body are range-restricted. Let r be a rule and P a class.

(1) Each variable of type P is range-restricted in r.
(2) If all variables in t_1 are range-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 range-restricted.

Such restrictions are not sufficient to obtain languages in PTIME. Invention must also be controlled as illustrated by Example 3.2: A program is *invention-free* if no variable occurs in the head and not the body of a rule. "Invention-freedom" is too drastic, it disallows new oids. To control invention, we use "recursion-freedom."

Let Γ be a program such that (*) the leftmost symbol of each rule is a relation name.[6] Γ is *recursion-free* if the directed graph $G(\Gamma)$ is acyclic, where: 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) (a) n is a relation or class name occurring in *body*(r), or
    (b) n is a class name and for some variable x occurring in *body*(r), n appears in the type of x;
(2) (a) n' is the leftmost symbol of r, or

[6] This technical restriction is imposed to simplify the presentation. The extension to the general case is a straightforward, although tedious task.

(b) n' is a class name and some variable x of type n' occurs in $head(r)$ and not in $body(r)$.

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

Definition 5.3. An (IQL^{pr}) IQL^{rr} program Γ is an IQL program if it is of the form $\Gamma_1; \ldots; \Gamma_k$ where for each i in $1 \cdots k$, Γ_i is (ptime) range-restricted and is either recursion-free or invention-free.

Clearly, IQL$^{rr} \subset$ IQL^{pr} by Definitions 5.1 and 5.2.

We could have chosen more involved criteria. The rationale for our definitions is that they are simple, subsume most popular query languages and suffice to show the principal result of this section.

THEOREM 5.4. *Each query expressed by an IQL^{pr} program can be answered in time polynomial in the size of the input instance.*

To prove this theorem, three lemmas are used: one deals with ptime-restricted programs; the other two treat respectively the recursion-free and the invention-free cases.

We first show that given an input database I and a ptime-restricted program Γ, the image by $\gamma_1(\Gamma)$ of I can be computed in polynomial time with respect to the size of I. The computation that we consider is the computation by naive evaluation.

LEMMA 5.5. *Let Γ be a ptime-restricted program and I an instance. Then some J such that (I, J) is in $\gamma_1(\Gamma)$ can be computed in time polynomial in the size of I.*

PROOF. Let r be a rule in Γ. We *claim* that:

The set of valuations θ of the variables in r such that $I \models \theta body(r)$ can be computed in time polynomial in the size of I.

Suppose that this is the case. Then it is easy to see that the valuation-domain and one map in the valuation-map can be computed in polynomial time, so an instance J satisfying the conditions of the lemma can be constructed.

The proof of the claim is by induction on the *rank* of the variables in r. For each variable x in r, if x is ptime-restricted by Condition (1) of Definition 5.1, then $rank(x) = 0$. Now consider a shortest proof of ptime-restriction of the variable x in r (based on Definition 5.1): $rank(x)$ is the maximum rank of the variables used to show that x is ptime-restricted plus one.

The range of each variable can be constructed in PTIME by induction on variable rank k. For $k = 0$, the claim is obvious. Now suppose that it is true for k and that x is of rank $k + 1$. Suppose (the other cases are treated similarly) that x occurs in some term t_2, $t_1 = t_2$ is in $body(r)$, and all variables in t_1 are of rank less or equal to k. The range for t_1, that is, a set X of o-values, can be constructed in PTIME. Now by matching t_2 with each o-value in X, one can construct a range for x in PTIME. □

The second lemma deals with recursion-free programs.

LEMMA 5.6. *Let* Γ *be a recursion-free, ptime-restricted program and* I *an instance. Then some* J *such that* (I, J) *is in* $\gamma_\infty(\Gamma)$ *can be computed in time polynomial in the size of* I.

PROOF. Let j be the length of the longest path in $G(\Gamma)$. For each relation or class name n, let $order(n)$ denote the length of the longest path leading to n plus one. (This is well defined by acyclicity.) We show by induction that

(+) for each sequence $\{I_i = (\rho_i, \pi_i, \nu_i)\}$ with $(I_i, I_{i+1}) \in \gamma_1(\Gamma)$ for each i,

—for each R, $k \geq order(R)$, $\rho_k(R) = \rho_{order(R)}(R)$.
—for each P, $k \geq order(P)$, $\pi_k(P) = \pi_{order(R)}(P)$ and $\nu_k(\pi_k(P)) = \nu_{order(P)}(\pi_{order(P)}(P))$.

For suppose that (+) holds. Then, $\gamma_\infty(\Gamma) = \gamma_1^{j+1}(\Gamma)$ and Lemma 5.5 suffices to conclude.

Note first that, by (*), $\nu_i = \nu_1$ for each i, so only ρ_i, π_i vary. The proof is by induction on the order of nodes.

Basis. Let R be a relation name with $order(R) = 1$. Suppose that R is the leftmost symbol of some rule r; that is, r is a rule that may contribute to the modification of the extension of R. Consider the sets

$$\Xi_i = \{\theta \mid \theta \text{ a valuation of the variables in } body(r) \text{ and } I_i \vDash \theta body(r)\}.$$

Since $order(R) = 1$, and R satisfies (2-a), there is no symbol in $body(r)$ satisfying (1-a) or (1-b). Therefore, $body(r)$ contains no relation or class name, and the type of each variable in $body(r)$ is free of any class name. It is now easy to see that $\Xi_i = \Xi_1$ for each i. Thus, the sets

$$\Xi_i' = \{\theta \mid (r, \theta) \in val\text{-}dom(I_i)\}$$

are empty for $i > 1$. Hence, r is not used to derive a new fact after the first step.

Similarly, consider a class P with $order(P) = 1$. The extension of P can only be modified by rules r where some variable of type P occurs in the head and not the body of r. In such cases, P satisfy (2-b). Like in the relational case, one can show that such rules saturate after one step since $order(P) = 1$, which concludes the proof of the basis.

Induction. Suppose that (+) holds for each symbol of order less or equal to k. Let n be a symbol of order $k + 1$ and r a rule which, potentially, modifies the extension of n. By an argument similar to that used in the basis, the sets

$$\Xi_i = \{\theta \mid \theta \text{ is a valuation of the variables in } body(r) \text{ and } I_i \vDash \theta body(r)\}$$

is constant for $i > k + 1$, and the set

$$\Xi_i' = \{\theta \mid (r, \theta) \in val\text{-}dom(I_i)\}$$

are empty for $i > k + 1$. Hence r is not used to derive a new fact after step $k + 1$. Thus (+) holds for $k + 1$ and by induction, (+) holds. □

The third lemma deals with invention-free programs. To prove it, we need the auxiliary concept of branching factor. The set *o-values*(I), for instance I, is the

set of v such that v occurs in *ground-facts*(I) in one of the following ways: $R(\ldots, v, \ldots)$, $\hat{o} = v$, or $\hat{o}(v)$ for some R or some o. The *branching factor* of an o-value v is the maximum outdegree of a node in the finite tree representing v. The *branching-factor* of I is the maximum branching factor of an element in *o-values*(I).

LEMMA 5.7. *Let* Γ *be an invention-free, ptime-restricted program and* I *an instance. Then some* J *such that* (I, J) *is in* $\gamma_\infty(\Gamma)$ *can be computed in time polynomial in the size of* I.

PROOF. We will show the following two facts:

(1) Let S be a schema. Then there is a polynomial pol_S such that for each set C of constants and oids and for each integer m, the size of each instance J over S with constants and oids in C, and with branching factor $k \leq m$ is less than $pol_S(\#(C), m)$.
(2) Let n be the branching factor of I and m the maximum number of symbols in a rule of Γ. Then the branching factor of some image J of I is less than $\text{Max}\{m, n\}$.

For suppose that (1) and (2) hold. Then, the size of J is a polynomial in the size of I. Thus, the number of facts in $J - I$ is a polynomial in I, so there is a polynomial bound on the number of iterations of the one-step operator. By Lemma 5.5, the computation of some J is in PTIME.

First consider (1). The proof is by induction on the depth of the types of S and uses the fact that although the powerset is exponential, the powerset with bounded set size is polynomial. Now consider (2). The proof is by induction on the steps of the iteration. The argument for the basis $k = 1$ and the induction are similar.

Induction Hypothesis. Suppose that for some k, for each J_k such that $(I, J_k) \in \gamma_1^k(\Gamma)$, J_k has a branching factor less than $\text{Max}\{m, n\}$.

Let J_{k+1} be an instance with $(J_k, J_{k+1}) \in \gamma_1(\Gamma)$ and v in *o-value*(J_{k+1}). If v is in J_k, its branching factor is less than $\text{Max}\{m, n\}$ by the hypothesis. Otherwise, v is inferred by some rule in Γ. Consider a node in the tree v and the subtree rooted at that node. Two cases occur:

—a similar subtree occurs in J_k, so the branching factor of n is less than $\text{Max}\{m, n\}$, by the hypothesis.

—the node corresponds to an o-value construction of the rule, so the branching factor of n is less than m, by construction.

Thus, J_{k+1} has branching factor less than $\text{Max}\{m, n\}$. This completes the proof of the lemma. □

Theorem 5.4 follows directly from Lemmas 5.6 and 5.7. It also implies that IQL^{rr}, which is a practical and implementable fragment of IQL, has PTIME data-complexity.

6. *Type Inheritance*

In this section, we add type inheritance to our object-based data model. First, we need to reconsider the notions of oid assignment and of types.

To model inheritance, one would like to consider nondisjoint oid assignments and to regard as equivalent the two type expressions:

$$[A_1\colon D, A_2\colon D] \wedge [A_2\colon D, A_3\colon D] \text{ and } [A_1\colon D, A_2\colon D, A_3\colon D].$$

The intuitive reason is that, if $[A_1\colon D, A_2\colon D]$ (respectively, $[A_2\colon D, A_3\colon D]$) were to describe all the records with *at least* A_1, A_2 (respectively, A_2, A_3) fields, then the intersection would be all the records with at least A_1, A_2, A_3 fields. Unfortunately, under the semantics previously defined $[A_1\colon D, A_2\colon D] \wedge [A_2\colon D, A_3\colon D]$ is equivalent to $\emptyset$.

This example motivates the definition of the alternative $*$-*interpretations* $[\![\cdot]\!]_{\pi *}$, which express the desired equivalences [Cardelli 1988]. In all cases, replace $[\![\cdot\cdots]\!]_\pi$ by $[\![\cdot\cdots]\!]_{\pi *}$ except for tuples where,

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

Observe that, in this definition, $v_{k+1}, \ldots, v_l$ are o-values of totally unconstrained types.

Like before: 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. One can also show an analog to Proposition 2.2.1:

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

Let us now extend the definition of schema as follows:

Definition 6.2. A schema S is a quadruple $(\mathbf{R}, \mathbf{P}, \mathbf{T}, \leq)$, 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 *type-exp*($\mathbf{P}$) 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. As we shall see, it is possible to express this intended meaning of *isa* statements, by schemas and instances without *isa*. The main reason is the presence 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*. To understand *isa*, we relax this assumption. We have already encountered some of the typing problems caused by nondisjoint oid assignments (see Example 4.1.2). We must resolve these problems while still being able to type check programs. This leads us to introduce *inherited oid assignments*, which can be described using union types. As

a second step, we add type inheritance, via the $*$-interpretation of types and inherited oid assignments.

A consequence is that, in the presence of type inheritance, we can use IQL in order to express all computable database queries. Our approach is intended to show the existence, in the presence of inheritance, of a typable and complete database query language. This is interesting, because it is not obvious that typing, inheritance, and completeness are all compatible notions.

Also, our precise definition of type inheritance allows the precise formulation of a number of important, new database query language problems. What happens to typing, inheritance, and completeness if union types are not provided? Practical languages make direct use of inheritance, by allowing some forms of value coercions. What is a good coercion strategy, that does not trade off typing or completeness?

6.1. Inherited Oid Assignments. To preserve type checking, it seems necessary to know a priori what type of result we get at evaluation time, when we evaluate terms such as $\hat{x}$ (see Example 4.2). This means static knowledge of which classes oids can belong to. We formalize this static knowledge using a common engineering intuition: "oids are created in a single class and automatically belong to the ancestors of this class in the *isa* hierarchy."

Definition 6.1.1. 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:

$$\bar{\pi}(P) = \bigcup \{\pi(P') \mid P' \in \mathbf{P}, P' \leq P\} (\text{for each } P \text{ in } \mathbf{P}).$$

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

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 $[A_1\colon student, A_2; instructor]$, 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 2.3.2; in Conditions (1) and (2) of Definition 2.3.2, 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, for example, replacing *student* by *student* $\vee$ *ta*. With the modified instance definition and using union types, querying is reduced to the case of disjoint oid assignments, and IQL can be used directly.

6.2. On the *-Interpretation of Types. As mentioned above, 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 the $*$-interpretation of types from Section 2. Let us come back to the canonical example.

Example 6.2.1. A more succinct specification of the schema of Example 6.1.2 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 6.1.2.

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 6.1.2 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 **P**. We construct τ_P such that:

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

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

Definition 6.2.2. An instance I of schema (**R, P, T,** $\leq$) 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 oids $\cup \{\pi(P) \mid P \in \mathbf{P}\}$ 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 **P**).
(3) ν is total on $\pi(P)$, for each $P \in \mathbf{P}$ with $\mathbf{T}(P) = \{\tau\}$.

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

We insist on the use of unstarred interpretations here to have the schema fully specify the structure of o-values in legal instances.

Instance I of S = (**R, P, T,** $\leq$), in Definition 6.2.2, is also an instance of a schema without *isa*'s S' = (**R, P, T′**). The new types **T′** are obtained from **T** by: first using τ_P (as defined above) instead of $\mathbf{T}(P)$ and then replacing every occurrence of P by the disjunction of its "smaller-or-equal" classes (in $\leq$). The language IQL can now be used with no modification.

Remark 6.2.3. In our treatment of inheritance, we have made two important design choices: (1) the oid assignments are inherited, and (2) the value of an object has exactly the type specified by the *least* class in the *isa* hierarchy, where it belongs. We believe that both design choices are natural restrictions to impose. Clearly, they can be enforced by constraining the creation of oids in a precise class with the exact type of that class (e.g., this form of creation is implicitly enforced in the implementation of Bancilhon et al. [1988]).

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 **P** and *v-type-exp*(**P**). The set *v-type-exp*(**P**) consists of all type expressions in *type-exp*(**P**) constructed without $\vee$, $\wedge$, $\emptyset$; that is, we assume only base, finite set, and tuple construction. Clearly, this is too restrictive for practical applications. But, this is sufficient to allow us to focus of the essence of the difference between a value-based and an object-based model.

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

We only consider here 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.

7.1. PURE VALUES. 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 oids

occur in them, (2) they might have infinite depth. These *infinite trees* are variants of the infinite trees in Courcelle [1983]. The only difference is that set nodes do not have a fixed arity and the order of their children is not significant, whereas all functions in Courcelle [1983] 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: properties of the infinite trees in Courcelle [1983] 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 **P**, a *finite assignment* I for **P** is a function from **P** to *finite* sets of pure values. Each finite assignment I defines a function from *v-type-exp*(**P**) to sets of pure values, called the type *interpretation* given I.

The function $[\![\cdot]\!]_I$ is analogous to $[\![\cdot]\!]_\pi$ of Section 2. More precisely, for each P in **P,** $[\![\mathbf{P}]\!]_\mathbf{I} = \mathbf{I(P)}$, and $[\![\cdot]\!]_I$ extends to *v-type-exp*(**P**) by structural induction.

Definition 7.1.1. A *v-schema* S is a pair (**P, T**), where **P** is a finite set of class names and **T** is a function from **P** to *v-type-exp*(**P**) such that:

$$(+) \text{ for each } P \in \mathbf{P}, \mathbf{T}(P) \text{ is not a class name.}$$

Definition 7.1.2. A *v-instance* I over (**P, T**) is a finite assignment for **P** such that:

$$I(P) \subseteq [\![\mathbf{T}(P)]\!]_I \text{ for each } P \in \mathbf{P}.$$

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 **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 [Courcelle 1983]. An important consequence of the finiteness of assignments is that each value occurring in a v-instance is a regular tree.

PROPOSITION 7.1.3. *Each pure value occurring in a v-instance is a regular tree.*

PROOF. Let I be a v-instance. Let $\{v_1, \ldots, v_n\} = \bigcup I(P)$. By Definition 7.1.2, $\langle v_1, \ldots, v_n \rangle$ can be viewed as a solution of a system of equations $\{v_i = t_i(v_1, \ldots, v_n)\}$ where each t_i is a finite tree with leaves in $\{v_1, \ldots, v_n\}$. Since the sets are finite, we can view this system as an extended regular Greibach system [Courcelle 1983]. (The technical condition (+) is also imposed in these systems). By Courcelle [1983], the solution is unique and each component of the solution, that is, each v_i, is regular. □

Now let us compare object-based and value-based instances over schema (**P, T**).

From values to objects. Let I be a v-instance over (**P, T**). For each P in **P**, let f_P be a one-to-one mapping from $I(P)$ to O with $f_P(I(P)) \cap f_{P'}(I(P')) = \emptyset$ if $P \neq P'$. Let π be the oid assignment such that for each P, $\pi(P) = f_P(I(P))$. We have defined things so that for each P in **P** and o in $\pi(P)$, we have that $o = f_P(v)$ for some v in $I(P)$. By definition of v-instances, $I(P)$ is a subset of the interpretation of $\mathbf{T}(P)$ given I.

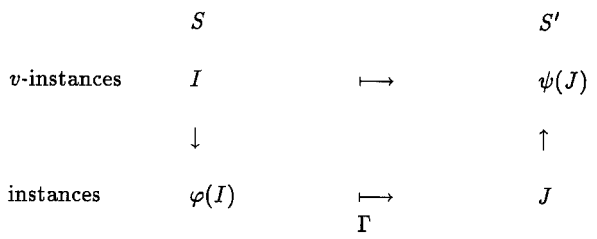

FIG. 2. Using IQL for the value-based model.

We next define the function ν. We can show by structural induction on types that, for each v in the interpretation of $\mathbf{T}(P)$ given I, there is a unique w_v in the interpretation of $\mathbf{T}(P)$ given π, such that: v is obtained from w_v by the simultaneous substitution

$$v = w_v[f_P^{-1}(o')/o' ; o' \in \pi(P), P \in \mathbf{P}].$$

Uniqueness is guaranteed, because the induction is constrained to one choice of type constructor at each step (union types would complicate things here). Let $\nu(o) = w_v$. It is easy to verify that (π, ν) is an instance over **(P, T)**. This instance clearly depends on the choice of the f_P. However, all the instances thereby obtained are O-isomorphic. Let us call $\varphi(I)$ one of these instances.

From objects to values. Let $I = (\pi, \nu)$ be an instance over **(P, T)** with ν defined for each oid occurring in π. Consider the set of equations

$$\{o = \nu(o) \mid \exists P \in \mathbf{P}, o \in \pi(P)\}.$$

The oids can be viewed as unknowns. Like in the proof of Proposition 7.1.3, the solution is unique. Let $o_1, \ldots, o_n$ be the oids in π, and $v_1, \ldots, v_n$ the solution of the system of equations. Let $\psi(I)$ be the v-instance defined by: $\psi(I)(P) = \{v_i \mid o_i \in \pi(P)\}$. Note that for o_i and o_j distinct, v_i and v_j may be the same (i.e., duplicates are eliminated).

We have defined translations of pure values into objects and vice-versa: φ can be thought of as producing o-values by adding oids and ψ as producing pure values by loosing oids. It is easy to show that these translations preserve information in the following sense:

PROPOSITION 7.1.4. *For each v-instance I,* $\psi(\varphi(I)) = I$.

Propositions 7.1.3 and 7.1.4 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'. Consider a v-instance I over S. Consider the mappings illustrated in Figure 2. Γ transforms $\varphi(I)$ into some instance 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. We shall also say that Γ transforms I into $\psi(J)$. 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 7.1.5. *IQL^v is vdio-complete*.

Remark 7.1.6. One can define the pure structure of a class as constrained by a given schema. To do that, one needs to use the regularity of the values in the class. An example of such an approach can be found in Ohori [1990].

8. *Conclusions*

We have extended the techniques of database theory in order to understand the concepts of "object-identity, types and type inheritance" in object-oriented databases. "Methods, method inheritance and encapsulation" are also important elements of an object-oriented database system and have parallels in programming languages, for example, *abstract types*. However, our techniques, with their emphasis on finite structures and *concrete* types, were not intended to deal with methods as programs or with the dynamics of method inheritance.

The language IQL is based on a logic programming paradigm. We believe that any typable language, based on a different paradigm but expressing the same database transformations, would have many fundamental similarities with IQL. In particular, the following features appear as central.

(1) *Types*. The concrete types used here are present in most object-oriented databases. Their type constructors largely determine what terms must be available in the language. Typing the language serves *both* for correctness and for efficiency. Efficiency is the main justification for the separation of database schema and instance, as well as for the requirement of type checking.

(2) *Basic query and update capabilities*. The means must be there to easily extract information from the database and modify its state. It should be easy to express common queries, for example, sets of conjunctive queries on complex-objects. It should be possible to both: find the *value-of-oids* and to *assign-values* to oids. Weak forms of assignment suffice for queries and insertions, but deletions introduce a certain amount of complexity.

(3) *Flow of control capabilities*. The means must be available for realizing sequential composition and looping. In rules, this can be provided by inflationary semantics and negation.

(4) *Cyclicity and oids*. There is some subtlety in the use of oids to code and manipulate cyclic structures, since these are represented in an acyclic manner by o-values. To deal with cyclicity, controlled use of typed pointers should be a key component of the language.

(5) *Invention of oids*. A primitive for oid invention must be in the language. We believe that this is necessary, if unbounded structures are to be constructed. Invention is a powerful mechanism; it is crucial if arbitrary computations are

to be simulated and it should be carefully restricted. Also, oid invention can be very useful for manipulating set types.

(6) *Relation vs. classes*. We believe that this dichotomy is essential for being able to express simple queries in a simple manner. We think that without this (in some sense) redundancy, the language will have difficulties in maintaining temporary results and eliminating duplicates, and it will use indirection excessively.

(7) *Incomplete information*. Incomplete information is important for a variety of database applications. The (benign) form of incomplete information that we use seems fundamental if complex cyclic structures have to be created in stages.

(8) *Type inheritance and coercions*. For union types, our language employs coercion and inheritance is handled indirectly, through union types. If a language is to use inheritance directly, it will need sophisticated coercion strategies. Finally, inheritance and union types need not be the only means to specify structure sharing. One might use other forms of polymorphism, for example, parametric types, abstraction over types.

To conclude this paper, it should be observed that the field of object databases has been quite active since IQL was first proposed. Most notably, as already mentioned, a standard model (ODL) and a standard query language (OQL) have been proposed by the ODMG [Cattell 1994]. The relative resemblance with the model used in IQL is an a-posteriori motivation for the present paper. The modelisation of data using objects and graph structures has become one of the major directions of research for modern databases notably influenced by the problem of accessing data on the World Wide Web. A number of languages have been proposed.[7] An important difference with the present work is that typing is in general much more flexible. (See Abiteboul [1997] for a survey on "semistructured data.") The extension of IQL to semistructured data and the study of complete languages in that new setting are interesting open issues.

ACKNOWLEDGMENTS. 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

IN MEMORIUM PARIS C. KANELLAKIS. *ACM Comput. Surv. 28*, 1 (Mar.), 5–15.

ABITEBOUL, S. 1997. Querying semistructured data. In *Proceedings of the International Conference on Database Theory* (invited paper) (Delphi, Greece).

ABITEBOUL, S., AND BEERI, C. 1988. On the power of languages for the manipulation of complex objects. INRIA Tech. Rep. No. 846.

ABITEBOUL, S., BEERI, C., GYSSENS, M., AND VAN GUCHT, D. 1987. An introduction to the completeness of languages for complex objects and nested relations. In *Nested Relations and Complex Objects*. Springer-Verlag, New York, pp. 117–138.

[7] See, for example, Abiteboul et al. [1996], Buneman et al. [1996], Christophides et al. [1994], Mendelzon and Wood [1995], Konopnicki and Shmueli [1995], and Mendelzon et al. [1996].

ABITEBOUL, S., AND GRUMBACH, S. 1988. COL: A logic-based languages for complex objects. In *Proceedings of EDBT*. pp. 271–293.

ABITEBOUL, S., GRUMBACH, S., VOISARD, A., AND WALLER, E. 1989. An extensible rule-based language with complex objects and data-formations. In *Proceedings of the DBPL-II Workshop* (Oregon). To appear.

ABITEBOUL, S., AND HULL, R. 1987. IFO: A formal semantic database model. *ACM Trans. Datab. Syst*. 12, 4 (Dec.), 525–565.

ABITEBOUL, S., AND HULL, R. 1988. Data functions, Datalog and negation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Chicago, Ill., June 1–3). ACM, New York, pp. 143–153.

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley, Reading, Mass.

ABITEBOUL, S., KANNELLAKIS, P., RAMASWAMY, S., AND WALER, E. 1995. Method schemas. *J. Comput. Syst. Sci. 51*, 3, 433–455.

ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WEINER, J. 1996/1998. The Lorel query language for semistructured data, 1996. Also in *J. Dig. Lib*., to appear. ftp://db.stanford. edu//pub/papers/lore196.ps.

ABITEBOUL, S., AND VIANU, V. 1987/1998. Transaction Languages for database update and specification. INRIA Tech. Rep. No. 715. Also in *J. Comput. Syst. Sci*., to appear.

ABITEBOUL, S., AND VIANU, V. 1988/1998. Datalog extensions for database updates and queries. INRIA Tech. Rep. No. 900. Also in *J. Comput. Syst. Sci.*, to appear.

ABITEBOUL, S., AND VIANU, V. 1989. Fixpoint extensions of first-order logic and datalog-like languages. In *Proceedings of the Symposium on Logic in Computer Science*.

ANDRIES, M., AND PAREDAENS, J. 1992. A language for generic graph transformations. In *Proceedings of the International Workshop WG 91*. Springer-Verlag, New York, pp. 63–74.

ATKINSON, M. D., AND BRUNEMAN, O. P. 1987. Types and persistence in database programming languages. *ACM Comput. Surv. 19*, 2 (June), 105–200.

BANCILHON, F. 1988. Object-oriented database systems. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., Mar. 21–23). ACM, New York, pp. 152–162.

BANCILHON, F., BARBEDETTE, G., BENZAKEN, V., DELOBEL, C., GAMERMAN, S., LECLUSE, C., PFEFFER, P., RICHARD, P., AND VELEZ, F. 1988. The design and implementation of O_2, and object-oriented database system. In *Proceedings of the OODBS2 Workshop* (Badmunster, RFA).

BANCILHON, F., CLUET, S., AND DELOBEL, C. 1989. Query languages for object-oriented database systems. In *Proceedings of the DBPL-II Workshop* (Oregon). To appear.

BANCILHON, F., DELOBEL, C., KANELLAKIS, P., EDS. 1992. *The Story of O_2*. Morgan-Kaufmann, San Mateo, Calif.

BANCILHON, F., AND KHOSHAFIAN, S. 1986. A calculus for complex objects. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Cambridge, Mass. Mar. 24–26). ACM, New York, pp. 53–60.

BANERJEE, J., CHOU, H.-T., GARZA, J. F., KIM, W., WOELK, D., BALLOU, N., AND KIM, H.-J. 1987. Data model issues for object-oriented applications. *ACM Trans. Inf. Syst. 5*, 1 (Jan.), 3–26.

BEERI, C. 1990. A formal approach to object-oriented databases. *J. Data Knowl. Eng. 5*, 4, 353–382.

BEERI, C., NAQVI, S., RAMAKRISHNAN, R., SHMUELI, O., AND TSUR, S. 1987. Sets and negation in a logic database language (LDL1). In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, Calif., Mar. 23–25). ACM, New York, pp. 21–37.

BREAZU-TANNEN, V., BUNEMAN, P., AND NAQVI, S. 1992. Structural recursion as a query language. In *Proceedings of the International Workshop on Database Programming Languages*. Morgan-Kaufmann, San Mateo, Calif., pp. 9–19.

BUNEMAN, P., DAVIDSON, S., HILLEBRAND, G., AND SUCIU, D. 1996. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Que., Canada, June 4–6). ACM, New York, pp. 505–516.

CARDELLI, L. 1988. A semantics of multiple inheritance. *Inf. Comput. 76*, 138–164.

CAREY, M. J., DEWITT, D. J., AND VANDENBERG, S. L. 1988. A data model and query language for EXODUS. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Ill., June 1–3). ACM, New York, pp. 413–423.

CATTELL, R. G. G. 1994. *The Object Database Standard: ODMG-93*. Morgan-Kaufmann, San Francisco, Calif.

CHANDRA, A., AND HAREL, D. 1980. Compatible queries for relational Data Bases. *J. Comput. Syst. Sci. 21*, 2, 156–178.

CHRISTOPHIDES, V., ABITEBOUL, S., CLUET, S., AND SCHOLL, M. 1994. From structured documents to novel query facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minn., May 24–27). ACM, New York, pp. 313–324.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June), 377–387.

CODD, E. F. 1979. Extending the database relational model to capture more meaning. *ACM Trans. Datab. Syst. 4*, 4 (Dec.), 397–434.

COURCELLE, B. 1983. Fundamental properties of infinite trees. *Theoret. Comput. Sci. 25*, 95–169.

DAHLAUS, E., AND MAKOWSKI, J. 1986. Computable directory queries. In *Proceedings of CAAP*. Lecture Notes in Computer Science, vol. 214. Springer-Verlag, New York, pp. 254–265.

DENNINGHOFF, K., AND VIANU, V. 1991. The power of methods with parallel semantics. In *Proceedings of the 17th International Conference on Very Large Data Bases*. pp. 221–232.

DENNINGHOFF, K., AND VIANU, V. 1993. Database method schemas and object creation. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Washington, D.C., May 25–28). ACM, New York, pp. 265–275.

FISHMAN, D. H., BEECH, D., CATE, H. P., CHOW, E. C., CONNORS, T., DAVIS, J. W., DERRETT, N., HOCH, C. G., KENT, W., LYNGBAEK, P., MAHBOD, B., NEIMAT, M. A., RYAN, T. A., AND SHEN, M. C. 1987. Iris: An object-oriented database management system. *ACM Trans. Inf. Syst. 5*, 1 (Jan.), 48–69.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk 80, the Language and Implementation*. Addison-Wesley, Reading, Mass.

HULL, R. 1986. Relative information capacity of simple relational schemata. *SIAM J. Comput. 15*, 3, 856–886.

HULL, R., AND SU, J. 1989. Untyped sets, invention and computable queries. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pa., Mar. 29–31). ACM, New York, pp. 347–359.

HULL, R., AND YOSHIKAWA, M. 1991. On the equivalence of database restructurings involving object identifiers. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Denver, Col., May 29–31). ACM, New York, pp. 328–340.

JAESCHKE, B., AND SCHEK, H.-J. 1982. Remarks on the algebra of non-first-normal-form relations. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Los Angeles, Calif., Mar. 29–31). ACM, New York, pp. 124–138.

KANELLAKIS, P. 1998. Elements of Relational Database Theory. In *Handbook of Theoretical Computer Science*, A. R. Meyers, M. Nivat, M. S. Paterson, P. Perrin, and J. van Leewen, eds. Elsevier-North Holland, Amsterdam, The Netherlands, pp. 1075–1144.

KHOSHAFIAN, S., AND COPELAND, G. 1986. Object identity. In *OOPSLA '86 Conference Proceedings* (Portland, Ore., Sept. 29–Oct. 2). ACM, New York, pp. 406–415.

KIFER, M., AND LAUSEN, G. 1989. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data* (Portland, Ore., May 31–June 2). ACM, New York, pp. 134–146.

KIFER, M., LAUSEN, G., AND WU, J. 1993. Logical foundations of object-oriented and frame-based languages. Tech. Rep. 93/06. Dept. Comput. Sci. SUNY Stony Brook, Stony Brook, N.Y.

KIFER, M., AND WU, J. 1989. A logic for object-oriented logic programming (Maier's O-logic: Revisited). In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pa., Mar. 29–31). ACM, New York, pp. 379–393.

KIM, W. 1988. A foundation for object-oriented databases. Tech. Rep. MCC, Austin, Tex.

KOLAITIS, P. G., AND PAPADIMITRIOU, C. H. 1988. Why not negation by fixpoint? In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., Mar. 21–23). ACM, New York, pp. 231–239.

KONOPNICKI, D., AND SHMUELI, O. 1995. W3QS: A query system for the World Wide Web. In *Proceedings of the 21st International Conference on Very Large Data Bases*. Morgan-Kaufmann, Zurich, Switzerland, pp. 54–65.

KORTH, H. F., ROTH, M. A., AND SILBERSCHATZ, A. 1985. Extended algebra and calculus for not 1NF relational databases. Tech. Rep. U. Texas at Austin, Austin, Tex.

KUPER, G. M. 1985. The logical data model: A new approach to database logic. Ph.D. dissertation. Stanford Univ., Stanford, Calif.

KUPER, G. M. 1987. Logic programming with sets. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, Calif., Mar. 23–25). ACM, New York, pp. 11–20.

KUPER, G. M., AND VARDI, M. Y. 1993. The logical data model. *ACM Trans. Datab. Syst. 18*, 379–413.

LÉCLUSE, C., AND RICHARD, P. 1989. Modeling complex structures in object-oriented databases. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pa., Mar. 29–31). ACM, New York, pp. 360–368.

LÉCLUSE, C., RICHARD, P., AND VELEZ, F. 1988. O_2, An object-oriented data model. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Ill., June 1–3). ACM, New York, pp. 424–434.

MAIER, D. 1986. A logic for objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming* (Washington, DC), pp. 6–26.

MAIER, D., OTIS, A., AND PURDY, A. 1985. Development of an object-oriented DBMS. *Quart. Bull. IEEE Datab. Eng. 8*.

MENDELZON, A., MIHAILA, G. A., AND MILO, T. 1996. Querying the World Wide Web. Draft. Available by ftp: milo@math.tau.ac.il.

MENDELZON, A. O., AND WOOD, P. T. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput. 24*, 6, 1235–1258.

NAQVI, S., AND KRISHNAMURTHY, R. 1988. Non-deterministic choice in Datalog. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*. Morgan-Kaufmann, Los Altos, Calif. pp. 416–424.

OHORI, A. 1990. Semantics of types for database objects ¡/B¿. *Theoret. Comput. Sci. 76*, 1, 53–91.

SCHEK, H., AND SCHOLL, M. 1986. The relational model with relation-valued attributes. *Inf. Syst.*

THOMAS, S. J., AND FISCHER, P. C. 1986. Nested relational structures. In *Advances in Computing Research*, vol. 3, *The Theory of Databases*. JAI press. Greenwich, Ct., pp. 269–308.

ULLMAN, J. D. 1987. Database theory—Past and future. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, Calif., Mar. 23–25). ACM, New York, pp. 1–10.

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems*, vol. I. Computer Science Press, New York.

VAN DEN BUSSCHE, J., AND PAREDAENS, J. 1991. The expressive power of structured values in pure OODB's. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Denver, Col., May 29–31). ACM, New York, pp. 291–299.

VAN DEN BUSSCHE, J., AND VAN GUCHT, D. 1992. Semi-determinism. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, Calif., June 2–4). ACM, New York, pp. 191–201.

VAN DEN BUSSCHE, J., VAN GUCHT, D., ANDRIES, M., AND GYSSENS, M. 1992. On the completeness of object-creating query languages. In *Proceedings of the IEEE Conference on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 372–379.

VERSO, J. 1986. Verso: A database machine based on non-1NF relations. INRIA Tech. Rep. (the Verson team). Also in *Nested Relations and Complex Objects*. Springer-Verlag. New York.

ZDONIK, S. 1985. Object management systems for design environment. *Quart. Bull. IEEE Datab. Eng. 8*, 4, 23–30.

RECEIVED NOVEMBER 1995; REVISED DECEMBER 1996; ACCEPTED MAY 1998