Object identify as a query language primitive

Serge Abiteboul, P.C. Kanellakis

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1022

Programme 4

OBJECT IDENTITY AS A QUERY LANGUAGE PRIMITIVE

Serge ABITEBOUL
Paris C. KANELLAKIS

Avril 1989

RR-1022

OBJECT IDENTITY AS A QUERY LANGUAGE PRIMITIVE*

Serge Abiteboul† I.N.R.I.A.

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

March 1989

Abstract

We demonstrate the power of object identities (oid's) as a database query language primitive. We develop an object-based data model, whose structural part generalizes most of the known complex-object data models: cyclicity is allowed in both its schemas and instances. Our main contribution is the operational part of the data model, the query language IQL, which uses oid's for three critical purposes: (1) to represent data-structures with sharing and cycles, (2) to manipulate sets and (3) to express any computable database query. IQL can be statically type checked, can be evaluated bottom-up and naturally generalizes most popular rule-based 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.

L'IDENTITE D'OBJET COMME PRIMITIVE DE LANGAGE DE REQUETES

Nous montrons la puissance de l'identité d'objet (oid) comme primitive de langage de requêtes. Nous développons un modèle de données orienté-objet dont la composante structurelle généralise la plupart des modèles de données complexes connus: les cycles sont admis à la fois dans le schéma et les données. Notre principale contribution réside dans la composante opératoire du modèle, le langage IQL, qui utilise les oids dans trois buts distincts: (1) pour représenter des structures avec partage d'objets et cycles, (2) pour manipuler des ensembles et (3) pour exprimer toutes les requêtes calculables. La vérification de types peut être réalisée de manière statique dans IQL. De plus, IQL peut être évalué en chaînage avant et généralise la plupart des langages base de données basés sur des règles. L'héritage de types est introduit dans le modèle sans rien changer au langage. Enfin, nous étudions un modèle analogue de données par valeurs dont la composante structurelle est basée sur des arbres réguliers infinis et dont la composante opératoire est encore le langage IQL.

---

*A preliminary version of this paper appears in the proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data.

†Work supported by the Projet de Recherche Coordonnée BD3.

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

1 Introduction

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

Object-oriented database systems will, most probably, be the next generation of commercial database systems (see [Ba88]). They are currently the focus of a great deal of experimentation and research, e.g., [MOP85,Zd85,Ba+87,F+87,CDV88,B+88]. These recent developments in databases are largely based on concepts and software tools from object-oriented programming, e.g., [GR83,Ba88,Ki88]. More generally, the integration of programming languages and database systems is an important research activity, for a detailed exposition of the state-of-the-art see [AtB87].

Unfortunately, much of the terminology currently used in object-oriented database systems is overloaded and experts disagree on the precise meaning of concepts such as: "object-identity, types, inheritance, methods and encapsulation etc." Consequently, there has been very little progress on understanding the "principles of object-oriented databases". This is in marked contrast with the previous generation of database systems, where the relational model of [Co70] provided the basis for many successful implementation efforts and, at the same time, for the development of an elegant and relevant theory [Ul88,Ka88].

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

Oid's have been part of many data models, for example they are called *surrogates* in [Co79], *l-values* in [KV84], or *object identifiers* in [AH87]. They have recently been highlighted as an essential part of object-oriented database systems [KC86]. A variety of reasons have been given for their use, e.g., structure sharing, updates [AH87] or the encoding of cyclicity [KV84]. We use oid's for the traditional encoding of directed (perhaps cyclic) graphs, but also for the manipulation of sets and for making our query language fully expressive. At an intuitive level, oid's are "typed pointers" and IQL is based on a *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 [Co70], most complex-object data models, e.g., [AB87,TF86,JS82,KRS85,SS86,Ve86], and the logical data model (LDM) [KV84,Ku85]. It can be viewed as the common upper bound of the models used in [KV84,AB87]. The pleasant surprise is that little mathematical simplicity had to be traded-off in order to achieve this synthesis. The actual definitions are not much longer than those for the relational model.

The operational part of the data model, the language IQL, is also surprisingly simple both in syntax and semantics. It has three basic properties: (1) it is a rule-based, (2) it can be statically type checked, and (3) it is complete, in the sense that it expresses exactly all

database transformations with certain desirable properties. Let us comment on these three points: (1) highlights the declarative nature and mathematical clarity of the programming paradigm used, (2) illustrates what is controlled about the use of pointers, and (3) involves generalizing the basic theorem of [CH80] 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 [KV84,AB87,AG88] and corresponds to strong typing in programming languages. A number of recent language proposals in this area do not have these properties. For example, in [BK86,Ma86,KW89] there is no instance-schema separation and the query languages can be viewed as untyped extensions of Prolog.

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

Structural Part: An *instance* consists of "data" in the form of: (1) a finite set of *o-values*, i.e., values containing oid's, and (2) a *partial function* ν of oid's to o-values, this mapping is the essence of the data model. Oid's and constants are o-values, but so are finite trees built-out of constants and oid's via finite tuple or set constructors. We allow ν to be partial; this is in order to model incomplete information and will be very useful in the operational part of the model. Note that, repeated applications of ν on oid's 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 oid's, where these oid's are mapped through ν to o-values of the same type $\mathbf{T}(P)$. An important assumption is that the classes of any legal instance are pairwise disjoint sets of oid's.

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

The dichotomy between relations and classes is the only design decision that slightly complicates the structural part. Its justification is that it greatly simplifies the operational part. Since relations are sets of o-values, duplicates are eliminated from them at a logical level. Thus, it is possible to program directly in popular rule-based formalisms, e.g., Datalog. Relations can name subsets of classes and function as useful temporaries. Also, this distinction allows a direct generalization of both [KV84] and [AB87].

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:

$\mathbf{T}(\textit{1st-generation})$ = [name: *string*, spouse: *1st-generation*, children: {*2nd-generation*}]
$\mathbf{T}(\textit{2nd-generation})$ = [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, e.g., *string*, and to class names, e.g., *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 oid's. Classes are assigned disjoint sets of oid's. The partial function ν assigns o-values to the oid's of the instance. Each one of these oid's has a value of the right type or is undefined. So, again strictly speaking, the type of $\pi(P)$ is $\{P\}$ and the type of $\nu(\pi(P))$ is $\{\mathbf{T}(P)\}$.

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

π(*1st-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* }],
ν(*eve*) = [name: Eve, spouse: *adam*, children: { *cain, abel, seth, other* }]
ν(*cain*) = [name: Cain, occupations: { Farmer, Nomad, Artisan }],
ν(*abel*) = [name: Abel, occupations: { Shepherd }],
ν(*seth*) = [name: Seth, occupations: { }]
ν(*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 [AG88], for the manipulation of sets, and the detDL language of [AV88], for the invention of new oid's. The focus was on adding the minimum to Datalog rules in order to obtain an object-based language, that can express all computable queries.

In summary, IQL is inflationary Datalog with negation [AV88,KP88], combined with set/tuple types, invention of new oid's, 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. IQL is the first database language with this property. 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 oid's used in the output do not matter only their interrelationships do. Let us examine the computation in IQL in four stages:

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

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

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

In the second stage, we produce two oid's per node using a semantics in the style of detDL [AV88]. We use a relation R' with $\mathbf{T}(R') = [A_1 : D, A_2 : P, A_3 : P']$ whose tuples contain oid's from class P and from another class P'. Class P' is a class with $\mathbf{T}(P') = \{P\}$, i.e., it's oid's have values that are sets of oid's from P. The following rule invents two oid's for each node, one of which will go into class P and the other into class P'.

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

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

In the third stage, we nest the oid's representing nodes in P into sets of successors of a node. This nesting of elements q is done by using the oid's p' of P' as temporary names. Each p' is set valued and its value, noted $\widehat{p'}$, is a set in which the corresponding q's are collected. This dereferencing and assignment to objects in P' simulates the effect of a COL data-function [AG88] or a grouping in LDL [Be+87].

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

In the final stage, the nodes of P have been grouped into P', and the connection in R' between x, p, p' is used to produce the desired result. Note that the value of some node p is a tuple with the name of the node as first component, and a set of P-oid's as a second component. This weak form of assignment is performed only when $\hat{p}$ was undefined (see [AH88]). No further changes are made to $\hat{p}$.

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

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

An important primitive in the language is the invention of oid's. This serves a triple goal: (1) objects may be part of the result and oid's must be assigned to them, (2) invented oid's

are used for set manipulation, (3) they are also used to obtain completeness in the sense of [CH80]. 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 [Be+87], *data-functions* in COL [AG88], or *universal quantification* [Ku87]. Thus, one of our contributions is to show that: *the manipulation and creation of sets can be realized only using invented oid's.*

We examine (3) in detail in Section 4. The notion of completeness of [CH80] is adapted to our context. Intuitively, the language must capture all transformations that are recursively enumerable and that preserve some isomorphism properties [CH80,Hu86]. Completeness results have been shown for the relational [CH80,AV87,AV88] and for many complex-object data models [DM86,A+89a,HS88].

Our notion of completeness is more general than the notion used in [CH80,AV88]. However, on relational schemas, the two notions coincide. The originality of our extension comes from the presence of oid's: two instances are viewed as identical if they are isomorphic up-to renaming of oid's. A basic contribution is a completeness result for IQL. *For disjoint input-output schemas, we show that all database transformations are expressible in IQL, up to copy elimination.* In many cases we can express copy elimination in IQL, but it is open if this technical restriction is necessary. Disjoint input-output schemas are sufficient for the study of queries and updates, such as insertions. To obtain completeness for non-disjoint schemas, we need to add non-inflationary features to IQL. These are based on the study of deletions in [AV88].

In Section 5, we specialize IQL using a number of syntactic restrictions. This specialization allows us to discover as IQL *sublanguages most of the popular rule-based formalisms.* We also show that these restrictions can be used to guarantee efficient query evaluation, i.e., with PTIME *data-complexity.* In [A+89b], similar restrictions are used in the context of the COL language to obtain queries that are evaluable 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 static typability of IQL programs. However, the removal of this condition is necessary if one is to study *type inheritance* as proposed in [Ca88]. 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: Oid's can be viewed as a syntactic trick to avoid manipulating recursive objects. The same is true for the use of class names in the type syntax. Even with these devices, recursive structures stay in the background in a fundamental way. Object-based systems often allow features such as *equality-by-value*, which is a precise way of addressing the underlying infinite objects. We illustrate a natural connection with a *value-based model* founded on regular infinite trees [Co83]. Our analysis allows us to show that IQL can serve as a language for this model as well. Object identities, in this context, loose all semantic denotation to become purely, primitives of the language. This is a non-trivial link between value-based and object-based [Ul87]. A value-based point of view can be used to understand *pure-values* (no oid's), *pure types* (no classes in the type syntax) and *equality-by-value* (as a coercion mechanism for realizing inheritance).

A major motivation for our work was the study of the formal aspects of the O_2 system, [B+88]. O_2 is a multilanguage object-oriented database system. Its structural data model, see [LRV88,LR89], is a subset of the structural data model with inheritance described here. The prototype implementation presented in [B+88] does not have union types, but does have type inheritance. Inheritance in O_2 is constrained by (pure) type compatibility: this facilitates coercions and the use of inheritance in queries [LRV88]. The operational part of O_2 is based on general purpose programming languages, such as *C* or *Basic*, enriched with language independent statements for database objects. Features of IQL, such as strong typing, relations as temporaries, oid-invention, assign-to-oid and value-of-oid, are present in this language independent formalism. A stand-alone query language is also investigated in [BCD89].

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*). 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?" We conclude, in Section 8, with 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 *oid's* $O = \{o_1, o_2, \ldots\}$. Throughout our exposition, we use the generic notation $[A_1 : \cdots, \ldots, A_k : \cdots]$ (where k is a 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 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 2.2 Let $\mathbf{R}$ be a finite set of relation names and let $\mathbf{P}$ be a finite set of class names.

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

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

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 *types*($\mathbf{P}$), is given by the following abstract syntax, where τ is a type expression, P an element of $\mathbf{P}$ and $k \geq 0$:

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

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

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

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

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

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

Proposition 2.3 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:

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

are equivalent over all π, and they are equivalent over all disjoint π to $[A_1 : D, A_2 : \{\emptyset\}]$. 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 $\emptyset$. $[A_1 : \emptyset]$ and $\emptyset$ are equivalent but $\{\emptyset\}$ and $\emptyset$ are not.

The above semantics combined with disjoint oid assignments are not the only desirable and simple alternative. In particular, to model inheritance, one would like to consider nondisjoint oid assignments and to regard as equivalent the two type expressions:

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

The intuitive reason is that, if $[A_1 : D, A_2 : D]$ (respectively, $[A_2 : D, A_3 : D]$) 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 formally defined above $[A_1 : D, A_2 : D] \wedge [A_2 : D, A_3 : D]$ is equivalent to $\emptyset$.

This example motivates the definition of the alternative $*$-*interpretations* $[\![\cdot]\!]_{\pi *}$, which express the desired equivalences [Ca88]. In all cases, replace $[\![..]\!]_\pi$ by $[\![..]\!]_{\pi *}$ except for tuples where,

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

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

Like above: 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. Once again one can show:

Proposition 2.4 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. □

2.3 Database Schemas and Instances

In this subsection, we present schemas and instances and comment on their definitions.

Definition 2.5 A schema S is a triple $(\mathbf{R}, \mathbf{P}, \mathbf{T})$, where $\mathbf{R}$ is a finite set of relation names, $\mathbf{P}$ is a finite set of class names, and $\mathbf{T}$ is a function from $\mathbf{R} \cup \mathbf{P}$ to *types*$(\mathbf{P})$.

Definition 2.6 An *instance* I of schema $(\mathbf{R}, \mathbf{P}, \mathbf{T})$ is a triple (ρ, π, ν), where ρ is an o-value assignment for $\mathbf{R}$, π is a disjoint oid assignment for $\mathbf{P}$, and ν is a partial function from the set $\cup \{ \pi(P) \mid P \in \mathbf{P} \}$ of oid's to o-values, such that:

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

The partial function ν associates o-values to the oid's in the instance (i.e., the oid's 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 2.6, 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 2.6 and from the semantics of types.

Let $I=(\rho, \pi, \nu)$ be an instance of a schema $S=(\mathbf{R}, \mathbf{P}, \mathbf{T})$. 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 2.6, ν must be total for set valued oid's. Based on this, we follow the convention that: if for some set valued oid o, there is no ground fact $\hat{o}(v)$ then $\nu(o) = \{\}$, and if for some non-set valued oid o there is no ground fact $\hat{o} = v$ then ν is undefined at o.

Finally, we use the terminology: *instances(S)* for the set of all instances of schema S; *objects(I)* for the set of all oid's occurring in I; *constants(I)* for the set of all constants occurring in I. By Definitions 2.1 and 2.2, 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*[1] *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 depth 2. In Appendix B of [Ku85], 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: incomplete information can be modeled using oid's with undefined value. So the structural definitions have some capability for expressing incomplete data, even without the machinery of lattices on terms, e.g., in [BK86]. Besides this, there is an important technical reason for having oid's with undefined values. The language IQL builds objects in stages, and oid's with undefined values are used in the intermediate stages. For o-values of set type $\{\tau\}$, we use $\{\}$ in order to achieve the same effect without any ambiguity; this is the rationale for Condition (3) in the instance definition.

Remark: 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 [KV84,AB87] 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'.

[1] 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.

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 = (**R**,**P**,**T**), where terms, literals and rules are defined as follows.

Terms: Assume that there are pairwise disjoint, countably infinite sets of variables for each τ in *types*(**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 **R** is a term of type $\{\mathbf{T}(R)\}$ and each P in **P** is a term of type $\{P\}$,
- for each P in **P** and variable x of type P, $\hat{x}$ is a term of type $\mathbf{T}(P)$,
- for $t_1,...,t_k$ terms of type τ, $\{t_1,...,t_k\}$ is a term of type $\{\tau\}$,
- for $t_1,...,t_k$ terms of types $\tau_1,...\tau_k$, $[A_1{:}t_1,...,A_k{:}t_k]$ is a term of type $[A_1{:}\tau_1,...,A_k{:}\tau_k]$.

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

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

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

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

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

Remark: Terms, literals and rules as defined here are pretty much standard, with some important additions. These are: (1) the typing for $R, P, \hat{x}$ in terms, (2) the relationship of the syntax of heads or facts with the ground facts of an instance, (3) the more liberal typing of equality in the bodies that will 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 [CH80]. Constants can be added easily without changing the framework (see [AV88]).

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 [AV88]. They are slightly complicated by two aspects of the language: (1) the particular mechanism used for oid invention, and (2) the *weak* assignment of o-values to non-set valued oid's based on Condition (*) below.

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

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

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

Given a program Γ and an instance I, the *valuation-domain*, denoted *val-dom*(Γ,I), is defined as follows:

val-dom$(\Gamma,I) = \{\ (r,\theta) \mid r \in \Gamma, I \models \theta$*body*$(r)$,
θ is a valuation exactly on variables in *body*(r),
and there is *no* extension $\bar{\theta}$ of θ such that $I \models \bar{\theta}$ *head*$(r)\}$

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

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

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

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

Inflationary operators: Given 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-facts(J) = *ground-facts*(I) $\cup$ { $\eta(r,\theta)$*head*(r) | for some r,θ subject to (*) below }
$\cup$ { $P(o)$ | for some r,θ and x of type P, $o = \eta(r,\theta)x$ and o 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.

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 oid's. (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 4.3, 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 oid's. 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 oid's. 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 Static 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. Type checking might be performed *statically*, before the evaluation on any input, or *dynamically*, during the evaluation on some input.

In IQL, one goal of type checking is to guarantee the soundness of programs. In other words, type checking is used to guarantee that the result is a *correct* instance. Another goal of type checking in IQL is to increase the *efficiency of evaluation*, e.g., to decrease the size and the cost of computing the valuation-domain. This latter use is a major justification for the separate notions of schema and instance in data models: "the schema contains the type information that is used to make retrieval efficient".

We claim that IQL programs can be statically type-checked. (Given the informal definition of static type checking, this assertion has to be informal). 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 statically 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, statically deciding if this inexpensive check is needed in some evaluation is not recursive; see [AH88].

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

Continuing with relational schemas, other relational languages can be viewed as IQL sublanguages, for example *detDL* [AV88]. The differences between detDL and IQL restricted to relations are: slightly different semantics for valuation-domains and invented constants in detDL versus invented oid's in IQL. However, it is very simple to simulate detDL in IQL.

It is shown in [AV88] 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* [AG88,A+89b]. The next example shows the IQL realization. For better clarity, we use capital letters, e.g., X,Y, for set variables.

Example 3.1 Let $(\mathbf{R},\mathbf{P},\mathbf{T})$ 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, use the single rule:

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

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

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

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

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

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

and Γ_2:

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

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

It should be noted that no invention is required in COL to perform the nesting: it is realized using *data functions*. A data function can be viewed as a *parametrized relation*

and is therefore based on a more elaborate concept than the relations in IQL. However, data functions can be simulated in IQL using invented oid's. We chose here to have oid invention, since such a feature serves many other purposes as well in our context.

One can show that each COL query can be computed using an IQL program. The proof is easy given the above programs for nest/unnest. As a consequence, all *algebraic operations on complex objects* of [TF86,JS82,SS86,AB87], and the calculus queries of [AB87,KRS85] 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 [AB87] is *powerset*. This operation is expensive: it is exponential in the input size. Indeed, we will emphasize sublanguages of IQL that cannot express the powerset, but can express important classes of queries evaluable in time polynomial in the input instance size. The powerset operation is considered in the next example. This example will provide all the necessary guidelines for the restrictions that will be imposed on IQL to obtain efficiently evaluable sublanguages.

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

$R_1(X) \longleftarrow X = X$

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

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

The powerset program consists of the rules:

$R_1(\{\}) \longleftarrow$

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

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

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

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

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

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

Note that in this second way of computing the powerset, invention of oid's occurs in a "loop". Such recursion with invention of oid's may clearly be the cause of nonterminating computations. For instance, let R_3 be a relation with $\mathbf{T}(R_3) = [A_1{:}P,A_2{:}P]$. Then the rule

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

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

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

Example 3.3 Consider the two schemas:
S has only one class P with $\mathbf{T}(P) = P \vee [A_1 : P, A_2 : P]$ and
S' has only one class P' with $\mathbf{T}'(P') = [A_1 : \{P'\}, A_2 : \{[A_1 : P', A_2 : P']\}]$.
We use one temporary relation R with $\mathbf{T}(R)= [A_1 : P, A_2 : P']$ and omit the attributes, when there is no ambiguity.
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 oid's) 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:
$R(x, x') \longleftarrow P(x)$
$\widehat{x'} = [\{y'\}, \emptyset] \longleftarrow R(x, x'),\ R(y, y'),\ y = \widehat{x}$
$\widehat{x'} = [\emptyset, \{[y', z']\}] \longleftarrow R(x, x'),\ R(y, y'),\ R(z, z'),\ [y, z] = \widehat{x}.$

An "inverse" mapping from S' to S can be realized using the rules:
$R(x, x') \longleftarrow P'(x')$
$\widehat{x} = w \longleftarrow R(x, x'),\ R(y, y'),\ y = w,\ \widehat{x'} = [\{y'\}, \emptyset]$
$\widehat{x} = w \longleftarrow R(x, x'),\ R(y, y'),\ R(z, z'),\ [y, z] = w,\ \widehat{x'} = [\emptyset, \{[y', z']\}].$

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

4 On the Expressive Power of IQL

Only some binary relations on instances can be input-output transformations defined by database language programs, for example instances must be well-typed. To formalize what are the meaningful binary relations, we extend the notion of database (db-) transformation of [AV88] and, thus, the notion of computable relational query of [CH80].

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. In some cases we can express copy elimination in IQL, but it is open if this technical restriction is necessary.

Finally, we describe how to extend IQL in order to remove the disjoint input-output schema restriction.

4.1 Db-transformations

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

The following definition states the four conditions that a binary relation on instances should satisfy, in order to qualify as a db-transformation. The first three conditions are standard and capture *well-typedness*, *effective computability*, and *genericity*. The justification for genericity is that a query language should not "interpret" atomic elements, such as constants and oid's, see [CH80,Hu86]. The fourth condition is new and expresses a form of *functionality.*

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

1. $\exists\ S,S'$ such that $\gamma \subseteq$ *instances*$(S) \times$ *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,

if (I,J) is in a db-transformation γ then *constants*$(J) \subseteq$ *constants*(I).

In contrast, the kind of functionality enforced by Condition (4) allows the presence of oid's in the output that were not in the input. This is a significant addition to the frameworks of [AV88,CH80]. It is important to be able to create new oid's in the output, if one wishes to manipulate in a general fashion the types available in the data model.

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

We now prove a soundness theorem: 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 4.2 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 $\hat{x}(z) \longleftarrow$ ..., and a rule $\hat{y}(\{z,z'\}) \longleftarrow$ ..., 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. □

Theorem 4.3 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 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. By the previous example, disjointness is necessary.

Consider Condition (3). Let (I,J) be in γ, h be a DO-isomorphism, and $I_0=I,...,I_n=I'$, $J = I'[S]$ 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 $hI=hI_0,...,hI_n=hI'$, $hJ = hI'[S]$ 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 the input, 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 *non-erasing* TM. Following [AV87,AV88], we first consider only disjoint input-output schemas. These suffice for a general study of database *queries* and *insertions*.

4.2 Disjoint Input-Output Schemas: Queries and Insertions

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 oid's 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 [CH80]. 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 4.4 Each dio-transformation for relational schemas is the semantics of some IQL program.

Proof: The proof follows the proof of [AV88] 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 4.5 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. Oid's 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 4.4 is used to conclude. □

For dio-transformations, we use Proposition 4.5 to obtain completeness of IQL "up to copy elimination". We show that given a dio-transformation γ, there is an IQL program which on input I_0 constructs finitely many copies of images of I_0 through γ. These copies are guaranteed to be identical up to renaming of the oid's and are distinguishable from each other (but we don't know whether one of these copies can be selected by an IQL program).

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

Definition 4.6 Let S be a schema with classes $\{P_1, ...P_n\}$ and I an instance of S. We define $\bar{S}$, the *schema for copies* of S by augmenting S with a single new relation name $\bar{R}$ with associated type $\{P_1 \vee ... \vee P_n\}$. An instance $\bar{I}$ of $\bar{S}$ is an *instance with copies of* I if there are O-isomorphic copies $I_1, \ldots, I_n$ of I such that:

1. *ground-facts*$(\bar{I}[S])$ = *ground-facts*$(I_1) \cup \ldots \cup$ *ground-facts*(I_n),

2. $J(\bar{R}) = \{objects(I_1), ..., objects(I_n)\}$, where $objects(I_i)$ $(i = 1, \ldots, n)$ 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 4.7 Each dio-transformation is the semantics of some IQL program up to copy.

To prove this theorem, we use two lemmas:

Lemma 4.8 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 $S = (\{R_1, ..., R_n\}, \{P\}, \mathbf{T})$ 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

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

of type

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

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 4.5, 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)... The first part of $\bar{\Gamma}$ visits each pair of integers in the above order; this exhaustive enumeration can be done as in [CH80,AV87], by realizing counters over a unary alphabet. For a pair (i,j), this part first invents i oid's. Then it constructs a relation of elements of type τ representing the set $\mathcal{J}_i$ of all instances over S, that can be constructed using the i oid's 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 oid's 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 $\mathcal{J}_{i,j}$ of tuples in $\mathcal{J}_i$ representing instances that are images of I_0 by γ, validated by $\Gamma_{y/n}$ in j steps. Two cases occur:

1. $\mathcal{J}_{i,j}$ is empty. Then Γ increments (i,j) and does another iteration of the first part.

2. $\mathcal{J}_{i,j}$ is not empty. Then relation $\mathcal{J}_{i,j}$ 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 $\mathcal{J}_{i,j}$ 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 4.9 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 $\tau = \tau_1 \vee ... \vee \tau_m$ by $[A_1 : \{\tau_1\}, ..., A_m : \{\tau_m\}]$. Our intention is to represent an o-value v of type τ by tuple $[A_1 : \alpha_1, ..., A_m : \alpha_m]$, where for each i in $1 \ldots n$, 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 4.8, 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 3.3. 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 4.7: Let γ be in a dio-transformation with output schema S=(**R**,**P**,**T**) and where $\mathbf{P} = \{P_1, ..., P_n\}$. Let $R_1, ..., R_n$ be new relation names, for each R in **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. $\mathbf{R}' = \{R_1, ..., R_n\} \cup \{\hat{R} \mid R \in \mathbf{R}\}$

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. $\mathbf{T}'(P)=\tau_1 \vee \ldots \vee \tau_n$ where for each i in $1 \ldots n$,
   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. $\pi'(P) = \pi(P_1) \cup \ldots \cup \pi(P_n)$,

2. for each R in $\mathbf{R}$, $\rho'(\hat{R}) = \rho(R)$,

3. for each i in $1 \ldots n$, $\rho'(R_i) = \pi(P_i)$.

Assume that $(I_0,I) \in \gamma$. Like in Lemma 4.9, 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 4.9, 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. □

Open problem: is copy elimination expressible in IQL?

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

Proposition 4.10 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 [AB87] is expressible in IQL.

Proof: The important observation is that, because there are no oid's in the output, the copy elimination is automatic. More specifically, in the proof of Lemma 4.8, there is no P and all tuple representations are identical. □

Proposition 4.11 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 4.8. Since the output instance contains only oid's, we can enumerate instances instead of sets of instances like in Lemma 4.8. To do that, in step (i,j), the i oid's 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 oid's can be enumerated. Thus there is no need for copies. □

Proposition 4.12 Each query in the calculus/algebra of the Logical Data Model of [KV84] is the semantics of some IQL program.

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

Remark: As in [AV88], one can introduce a *nondeterministic* dio-transformations and a nondeterministic variant of IQL that expresses all of them. With nondeterminism, selection of one out of a set of copies is easy.

Remark: 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 [AV88]. Intuitively, the order on the finite set of constants of the input together with the order of creation of oid's allows the enumeration of instances.

4.3 Nondisjoint 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 [AV88]. 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 propositions, remarks and open questions of the previous subsection can be extended analogously, from disjoint to nondisjoint input-output schemas. We omit the formal treatment, since the additional techniques are well known, see [AV87,AV88].

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 [AH87]). From a practical standpoint, it requires more involved evaluation mechanisms, e.g., with reference counts or garbage collection.

5 On the Sublanguages of IQL

Queries can be written directly in IQL, they can be statically 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 [AB87] and COL with range-restriction [A+89b]. 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 $\text{IQL}^{rr} \subset \text{IQL}^{pr} \subset \text{IQL}$. These sublanguages have PTIME data-complexity. Also, all the above popular query formalisms are contained in IQL^{pr}. The containment is proper, because many queries on cyclic schemas (such as the graph example from the introduction) are IQL^{rr} expressible.

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).

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.

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 oid's. To control invention, we use "recursion-freedom".

Let Γ be a program such that (*) the leftmost symbol of each rule is a relation name[2]. 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]This technical restriction is imposed to simplify the presentation. The extension to the general case is a straightforward, although tedious task.

2. (a) n' is the leftmost symbol of r, or
   (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 $\Gamma_1;...;\Gamma_k$ where for each i in $1...k$, 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.

For 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, i.e., 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, i.e., r is a rule that may contribute to the modification of the extension of R. Consider the sets

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

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

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

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: $R(...,v,...)$, $\hat{o} = v$, 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 oid's and for each integer m, the size of each instance J over S with constants and oid's 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 that $\text{Max}\{m,n\}$.

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. Let us extend the definition of schema as follows:

Definition 6.1 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 *types*($\mathbf{P}$) and $\leq$ is a partial order on $\mathbf{P}$ called *isa hierarchy.*

A common use of type inheritance is to specify, by the addition of an *isa* hierarchy to the schema, that certain classes share certain structural properties. 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 4.2). We must resolve these problems, without resorting to dynamic type checking. 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 statically typable and complete database query language. This is interesting, because it is not obvious that static 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 static 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 static typing or completeness?

6.1 Inherited Oid Assignments

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

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

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

Example 6.3 Let our schema contain four classes $P_1 \equiv$ *person* with $\mathbf{T}(P_1) = [name{:}D]$, $P_2 \equiv$ *student* with $\mathbf{T}(P_2) = [name{:}D, course\text{-}taken{:}D]$, $P_3 \equiv$ *instructor*, with $\mathbf{T}(P_3) = [name{:}D, course\text{-}taught{:}D]$, and $P_4 \equiv$ *ta*, with $\mathbf{T}(P_4) = [name{:}D, course\text{-}taught{:}D, course\text{-}taken{:}D]$.

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

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

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

In the previous example, there is no restriction on the types of classes related via the *isa* relationship. Namely, *isa* and types are declared separately and independently. Given $\bar{\pi}$, the type interpretations (as defined in Section 2) determine what are the possible o-values. Thus, instances can be defined by a single modification of Definition 2.6: in Conditions (1) and (2) of Definition 2.6 use

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

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

Wherever $\bar{\pi}(P)$ appears in the modified definition it can be replaced by $\cup \{ \pi(P') \mid P' \leq P \}$. This corresponds to replacing, in the types, a class P by the disjunction of its "smaller-or-equal" classes, e.g., replacing *student* by *student* $\vee$ *ta*. 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 6.4 A more succinct specification of the schema of Example 6.3 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 2.4, we can eliminate the intersection and get the type expressions explicitly given in Example 6.3.

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 6.3 should have exactly type [*name*:D, *course-taught*:D, *course-taken*:D], and no other attributes.

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

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

Such a type expression τ_P exists by Proposition 2.4. Now we can put everything together in a new definition that gives meaning to *isa*'s.

Definition 6.5 An instance I of schema $(\mathbf{R}, \mathbf{P}, \mathbf{T}, \leq)$ is a triple (ρ, π, ν), where ρ is an o-value assignment for $\mathbf{R}$, π is a disjoint oid assignment for $\mathbf{P}$, and ν is a partial function from the set of oid's $\cup \{ \pi(P) \mid P \in \mathbf{P} \}$ to o-values, such that:

1. $\rho(R) \subseteq [\![\mathbf{T}(R)]\!]_{\bar{\pi}}$ (for each $R \in \mathbf{R}$),
2. $\nu(\pi(P)) \subseteq [\![\tau_P]\!]_{\bar{\pi}}$ (for each P in $\mathbf{P}$).
3. 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=($\mathbf{R}$, $\mathbf{P}$, $\mathbf{T}$, $\leq$), in Definition 6.5, is also an instance of a schema without *isa*'s S'=($\mathbf{R}$, $\mathbf{P}$, $\mathbf{T}'$). The new types $\mathbf{T}'$ are obtained from $\mathbf{T}$ 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: 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 oid's in a precise class with the exact type of that class (e.g., this form of creation is implicitly enforced in the implementation of [B+88]).

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-types*(**P**). The set *v-types*(**P**) consists of all type expressions in *types*(**P**) constructed without $\vee, \wedge, \emptyset$, i.e., we assume only base, finite set and tuple construction.

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 oid's occur in them, (2) they might have infinite depth. These *infinite trees* are variants of the infinite trees in [Co83]. 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 [Co83] 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 [Co83] 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-types*(**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**, $[\![P]\!]_I = I(P)$, and

$[\![\cdot]\!]_I$ extends to *v-types*(**P**) by structural induction.

Definition 7.1 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-types*(**P**) such that:

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

Definition 7.2 A *v-instance* I over (**P**,**T**) is a finite assignment for **P** such that:
$I(P) \subseteq [\![\mathbf{T}(P)]\!]_I$ for each $P \in \mathbf{P}$.

Note the simplicity of these definitions, that generalize complex-object data models by adding cyclicity to both schemas and instances. The technical condition on the **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 [Co83]. An important consequence of the finiteness of assignments is that each value occurring in a v-instance is a regular tree.

Proposition 7.3 Each pure value occurring in a v-instance is a regular tree.

Proof: Let I be a v-instance. Let $\{v_1,...,v_n\} = \cup\, I(P)$. By Definition 7.2, $\langle v_1,...,v_n\rangle$ can be viewed as a solution of a system of equations $\{\ v_i = t_i(v_1,...,v_n)\ \}$ where each t_i is a finite tree with leaves in $\{v_1,...,v_n\}$. Since the sets are finite, we can view this system as an extended regular Greibach system [Co83]. (The technical condition (+) is also imposed in these systems). By [Co83], the solution is unique and each component of the solution, i.e., 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.

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

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

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
$\{\ o = \nu(o)\ |\ \exists\ P \in \mathbf{P},\ o \in \pi(P)\ \}$. The oid's can be viewed as unknowns. Like in the proof of Proposition 7.3, the solution is unique. Let $o_1,...,o_n$ be the oid's in π, and $v_1,...,v_n$ the solution of the system of equations. Let $\psi(I)$ be the v-instance defined by: $\psi(I)(P) = \{\ v_i\ |\ o_i \in \pi(P)\ \}$. 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 oid's and ψ as producing pure values by loosing oid's. It is easy to show that these translations preserve information in the following sense:

Proposition 7.4 For each v-instance I, $\psi(\varphi(I)) = I$. □

Propositions 7.3 and 7.4 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.

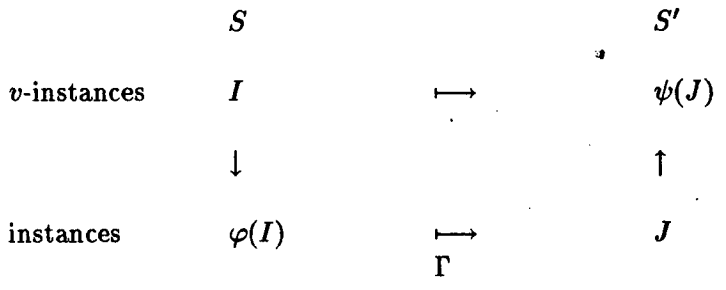

Figure 1: Using IQL for the value-based model

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 1. 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 7.5 IQL^v is vdio-complete. □

7.2 Pure Types

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

As mentioned in the previous section, values in a v-instance are regular trees. Their structure is constrained by the schema. A natural semantic definition for the *pure type* of a class P in a schema S would be $\{v \mid \exists$ instance I over S such that $v \in I(P)\}$, i.e., the set of all values that may be members of P. We next exhibit a constructive definition of a pure type and show that the two definitions are equivalent.

Example 7.6 Consider the following example:

$$\mathbf{T}(P_1) = [A_1{:}D, A_2{:}D],$$
$$\mathbf{T}(P_2) = [A_3{:}D, A_2{:}P_1], \text{ and}$$
$$\mathbf{T}(P_3) = [A_1{:}D, A_2{:}P_3].$$

Intuitively, the pure type of P_1 is given by $[A_1{:}D, A_2{:}D]$, and of P_2 by $[A_3{:}D, A_2{:}[A_1{:}D, A_2{:}D]]$.

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

Instead of introducing recursive syntax, it is simpler to define the pure type of P_3 using a sequence of finite types which are better and better approximations of P_3 [Co83]:

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

We first introduce the set $\omega\text{-}types(\mathbf{P})$ ($P \in \mathbf{P}$, $k \geq 0$) defined by:

$$\tau := D \mid P \mid \Omega \mid [\ A_1{:}\tau,\ldots,A_k{:}\tau\ ] \mid \{\tau\}.$$

The new element of course is Ω with interpretation the set $\{\perp\}$. Let σ be in $\omega\text{-}types(\emptyset)$. The *interpretation* of σ is given by the standard structural induction, and it does not depend on an interpretation of the P's. Note that if σ contains no Ω, it is a set of finite pure values.

We next introduce a partial order $\leq$ on the set of "finite trees with $\perp$" by: for each v, $\perp \leq v$, and if for each i, $v_i \leq w_i$, then $\{v_1,\ldots,v_n\} \leq \{w_1,\ldots,w_n\} \leq \{w_1,\ldots,w_n,\perp\}$ and $[A_1{:}v_1,\ldots,A_n{:}v_n] \leq [A_1{:}w_1,\ldots,A_n{:}w_n]$. Note that $\{d\} \equiv \{d,\perp\} \leq \{d,d'\}$, and thus, some increasing sequences do not have a least upperbound. This is easily fixed by considering only sequences with "bounded branching factor". A sequence of trees has *bounded branching factor* if for some integer k, each node in each tree of the sequence has less than k children. The following lemma is useful in order to constructively define pure types.

Lemma 7.7 Every increasing sequence with bounded branching factor has a least upperbound. $\square$

Let $S = (\mathbf{P},\mathbf{T})$ be a v-schema and P in $\mathbf{P}$. Let τ be in $\omega\text{-}types(\mathbf{P})$ and $\sigma(P)$ be in $\omega\text{-}types(\mathbf{P})$ for each P in $\mathbf{P}$. Then $\tau\ [\sigma(P)/P;P \text{ in } \mathbf{P}]$ is the ω-type expression obtained from τ by simultaneous substitution of $\sigma(P)$ for each P. Observe that for each P', $P'[\mathbf{T}(P)/P;P \text{ in } \mathbf{P}] = \mathbf{T}(P')$. Two such substitutions are of particular interest:

$$\mathbf{T}(\tau) = \tau\ [\mathbf{T}(P)/P;P \text{ in } \mathbf{P}] \text{ and } \mathbf{T}_\omega(\tau) = \tau\ [\Omega/P;P \text{ in } \mathbf{P}].$$

Example 7.8 Consider the schema $(\{P_1,P_2,P_3\},\mathbf{T})$ of Example 7.6. The following are in $\omega\text{-}types(\emptyset)$, and their interpretations do not depend on the P's.

$\mathbf{T}_\omega\ \mathbf{T}\ (P_3) = [A_1{:}D,A_2{:}\Omega]$, and
$\mathbf{T}_\omega\ \mathbf{T}^2\ (P_3) = [A_1{:}D,A_2{:}[A_1{:}D,A_2{:}\Omega]]$. $\square$

Based on the above notation and on Lemma 7.7 we have:

Definition 7.9 Let S = (P,T) and P be in $\mathbf{P}$. Define *pure-type*(S,P) as the set of pure values v such that: (1) v is regular and (2) v is the least upperbound of some increasing sequence $v_1,\ldots,v_i,\ldots$ of finite trees with $\perp$, where for each i, v_i is in the interpretation of $\mathbf{T}_\omega\mathbf{T}^i(P)$.

Proposition 7.10 Let $S = (\mathbf{P},\mathbf{T})$ and $P \in \mathbf{P}$ then,

$$\textit{pure-type}(S,P) = \{v \mid \exists \text{ instance } I \text{ over } S \text{ such that } v \in I(P)\}$$

Proof: ($\supseteq$) choose for v_i the best approximation of v in the interpretation of $\mathbf{T}_\omega\mathbf{T}^i(P)$. ($\subseteq$) By definition, v is regular, so *subtrees*(v) is finite. Let I be the instance defined by: for each P' in $\mathbf{P}$, $I(P') = \textit{subtrees}(v) \cap \textit{pure-type}(S,P')$. One can show that I is a v-instance of S. □

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, e.g., *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.

In the framework defined here, a number of interesting problems remain open. For example: (a) determine if copy elimination is expressible in IQL; (b) develop coercion strategies for inheritance that preserve static typability and expressibility; (c) analyze the power of inheritance in the absence of union types; (d) extend the value-based data model with union types; and (e) study coercion strategies that are based on pure values and pure types.

The language IQL is based on a logic programming paradigm. We believe that any statically 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 static 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, e.g., sets of conjunctive queries on complex-objects. It should be possible to both: find the *value-of-oid*'s and to *assign-values* to oid's. 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 oid's:** There is some subtlety in the use of oid's 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 oid's:** 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. **Relations vs classes:** 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, 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, e.g., parametric types, abstraction over types.

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

References

[AB87] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects, INRIA Technical report, No 846, 1988.

[AG88] S. Abiteboul and S. Grumbach. COL: a Logic-based Language for Complex Objects. In *Proc. EDBT*, 271–293, 1988.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM TODS*, 12:525–565, 1987.

[AH88] S. Abiteboul and R. Hull. Data-functions, Datalog and Negation. In *Proc. ACM SIGMOD*, 143–153, 1988.

[AtB87] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[AV87] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Update and Specification, to appear in *JCSS*. In *Proc. ACM PODS*, 260–268, 1987.

[AV88] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Language. In *Proc. ACM PODS*, 240–250, 1988.

[A+89a] S. Abiteboul and C. Beeri and M. Gyssens and D. van Gucht, An Introduction to the Completeness of Languages for Complex Objects and Nested Relations. To appear in Nested Relations and Complex Objects *Springer-Verlag*).

[A+89b] S. Abiteboul, S. Grumbach, A. Voisard, and E. Waller. An Extensible Rule-based Language with Complex Objects and Data-functions. In *Proc. DBPL-II Workshop*, Oregon USA, 1989. To appear.

[Ba88] F. Bancilhon. Object-Oriented Database Systems. In *Proc. ACM PODS*, 152–162, 1988.

[Ba+87] J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, and N. Ballou. Data Model Issues for Object-Oriented Applications. *ACM TOIS*, 5:1:3–26, 1987.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. Query Languages for Object-Oriented Database Systems. In *Proc. DBPL-II Workshop*, Oregon USA, 1989. To appear.

[Be+87] C. Beeri and al. Sets and Negation in a Logic Database Language (LDL1). In *Proc. ACM PODS*, 21–37, 1987.

[BK86] F. Bancilhon and S. Khoshafian. A Calculus for Complex Objects. In *Proc. ACM PODS*, 53–60, 1986.

[B+88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2, an Object-Oriented Database System. In *Proc. OODBS2 Workshop*, Badmunster RFA, 1988.

[Ca88] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[CDV88] M.J. Carey, D.J. Dewitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD*, 413–423, 1988.

[CH80] A. Chandra and D. Harel. Computable Queries for Relational Data Bases. *JCSS*, 21:2:156–178, 1980.

[Co70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.

[Co79] T. Codd. Extending the Database Relational Model to Capture more Meaning. *ACM TODS*, 4:4:397–434, 1979.

[Co83] B. Courcelle. Fundamental Properties of Infinite Trees. *TCS*, 25, 95–169, 1983.

[DM86] E. Dahlaus and J. Makowski. Computable Directory Queries. In *Proc. CAAP*, 1986. LNCS 214, Springer-Verlag.

[F+87] D. Fishman et al. Iris: an Object-Oriented Database Management System. *ACM TOIS*, 5:1:46–69, 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80, the Language and Implementation.* Addison-Wesley, 1983.

[Hu86] R. Hull. Relative Information Capacity of Simple Relational Schemata. *Siam J. of Computing*, 15:3, 1986.

[HS88] R. Hull and J. Su. Untyped Sets, Invention and Computable Queries. In *Proc. ACM PODS* 1989. To appear.

[JS82] B. Jaeschke and H.J. Schek. Remarks on the Algebra of Non-first-normal-form Relations. In *Proc. ACM PODS*, 124–138, 1982.

[Ka88] P. Kanellakis. *Elements of Relational Database Theory.* Brown U. Technical Report, 1988. To appear as a chapter in the Handbook of Theoretical Computer Science.

[KC86] S. Khoshafian and G. Copeland. Object Identity. In *Proc. OOPSALA*, 1986.

[Ki88] W. Kim. *A Foundation for Object-Oriented Databases.* Technical Report, MCC, 1988.

[KP88] P.G. Kolaitis and C.H. Papadimitriou. Why not Negation by Fixpoint? In *Proc. ACM PODS*, 231–239, 1988.

[KRS85] H.F. Korth, M.A. Roth, and A. Silberschatz. *Extended Algebra and Calculus for not 1NF Relational Databases.* U. Texas Austin Technical Report, 1985.

[Ku85] G.M. Kuper. *The Logical Data Model: a New Approach to Database Logic.* Stanford U., PhD thesis, 1985.

[Ku87] G.M. Kuper. Logic Programming with Sets. In *Proc. ACM PODS*, 11-20, 1987.

[KV84] G.M. Kuper and M.Y. Vardi. A New Approach to Database Logic. In *Proc. ACM PODS*, 86–96, 1984.

[KW89] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proc. ACM PODS*, 1989. To appear.

[LR89] C. Lecluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. ACM PODS*, 1989. To appear.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proc. ACM SIGMOD*, 424-434, 1988.

[Ma86] D. Maier. A Logic for Objects. In *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, Washington USA, 1986.

[MOP85] D. Maier, A. Otis, and A. Purdy. Development of an Object-Oriented Dbms. *Quarterly Bulletin of IEEE on Database Engineering*, 8, 1985.

[SS86] H. Schek and M. Scholl. The Relational Model with Relation-valued Attributes. *Information Systems*, 1986.

[TF86] S.J. Thomas and P.C. Fischer. Nested Relational Structures. In *Advances in Computing Research, Vol. 3, the Theory of Databases*, JAI press, 269–308, 1986.

[Ul87] J.D. Ullman. Database Theory - Past and Future. In *Proc. ACM PODS*, 1–10, 1987.

[Ul88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[Ve86] J. Verso. *Verso: a Database Machine Based on non-1NF Relations.* INRIA Technical Report, 1986. (Verso is a pen name for the Verso team). To appear in Nested Relations and Complex Objects *Springer-Verlag.*

[Zd85] S. Zdonik. Object Management Systems for Design Environments. *Quarterly Bulletin of IEEE on Database Engineering*, 8, 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique