JUNE 1991, Volume 14, No. 2

a quarterly bulletin of the
IEEE Computer Society
technical committee on

Data Engineering

CONTENTS

SPECIAL ISSUE ON FOUNDATIONS OF OBJECT–ORIENTED DATABASE SYSTEMS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE

1951-1991
40 YEARS OF SERVICE
IEEE COMPUTER SOCIETY

June 1991, Volume 14, No. 2

Data Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Data Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unreformed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Data Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Data Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non–member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the Issue Editor

Theoretical Foundations of Object-Oriented Database Systems

Object-orientation has emerged as a major theme in current work on database systems, and several research prototypes and commercial database products based on object-oriented paradigm are in different stages of development. The enthusiasm and energy devoted to the development of object-oriented database systems match, if not exceed, the development effort spent on relational systems in the late seventies and early eighties. Interestingly, however, the development of object-oriented database systems has taken a very different evolutionary path. While the relational systems started with a strong theoretical foundation, there is no consensus yet on a formal theory for object-oriented database systems. Experimental systems and products seem to be driving this field at this stage.

This issue presents a *sampling* of *some* recent attempts to provide a theoretical foundation for object-oriented database systems. The issue contains 12 papers. These papers cover various aspects of object-oriented database systems, including modeling, schema evolution, integrity constraints, views, and queries.

Abiteboul and Kanellakis point out the two facets of object-oriented data models — *structural* and *behavioral* — that reflect the two origins of object-oriented database systems — relational database theory and object-oriented programming. They formalize and analyze these two facets and give examples of their integration.

Beeri argues for the extension of the existing logic-based approaches to databases and programming languages for modeling object-oriented database systems. He presents an initial model approach that unifies the theories of relational databases, deductive programming, and abstract data types. He also argues for functions as first class values and a flexible function definition facility to model behavioral aspects of object-oriented database systems.

Kifer presents the salient features of F-logic that make it possible to provide a full logical formalization of object-oriented languages. F-logic breaks the distinction between classes, objects, and attributes which allows queries that may return sets of attributes, classes, or any other aggregation that involves these higher-order entities. It is also possible to define parametric classes in F-logic.

Object identity is a central concept in object-oriented database systems. Hull, Widjojo, Wile, and Yoshikawa differentiate between object identities and values, describe a formal model which encompasses object identities and values, and examine the impact of object identity in the contexts of data structuring and merging.

Breazu-Tannen, Buneman, and Ohori argue that the object-oriented database systems can be best understood in the framework of typed languages. They address the demands placed on programming languages by the addition of operations on records and "bulk" data types such as sets.

Lecluse and Richard describe the foundations of the O_2 database system. The O_2 data model differentiates between values and objects, and between types and classes. It also supports the notions of the consistency of a class hierarchy and database schema.

Coen-Porisini, Lavazza, and Zicari address the problem of schema evaluation in object-oriented database systems. They differentiate between structural and behavioral consistency of a schema and outline their solutions for maintaining schema consistency in the presence of schema modifications.

Object-oriented data models give rise to additional constraints beyond those meaningful under the relational model. Kim, Lee, and Seo present a framework for classifying integrity constraints in the context of an object-oriented data model on the basis of their performance impact.

Scholl and Schek describe how views may be supported in object-oriented database systems. They introduce object-preserving query semantics of a generic object-oriented query language and show how views defined by such query expressions may be updated. Dynamic reclassification allows objects to gain and loose types.

Vandenberg and DeWitt describe the algebraic fundamentals underlying the processing and optimization of EXCESS queries in the EXTRA/EXCESS DBMS. They describe the algebraic structures and their operators, the algebra's expressive power, and the algebraic query optimization.

Zdonik and Mitchell present the ENCORE data model and its query algebra, called EQUAL. This algebra generalizes relational operations by providing the capability to access and produce encapsulated, logically complex objects.

Finally, Daniels, Graefe, Keller, Maier, Schmidt, and Vance discuss query optimization in object-oriented databases in the context of Revelation project. They describe modeling features that support user-defined data types, consider their impact on query optimization, and discuss the Revelation approach to these problems.

Before closing, I would like to thank the authors for providing excellent papers at a short notice. It is my fond hope that this issue will provide impetus for further research in this important area.

Rakesh Agrawal
IBM Almaden Research Center
San Jose, California 95120

The Two Facets of Object-Oriented Data Models

SERGE ABITEBOUL* PARIS C. KANELLAKIS†

0. Introduction: Object-oriented database systems (OODBs) are new software systems built using techniques from databases, object-oriented languages, programming environments and user interfaces; for examples of this emerging technology see the edited collections [KL 89, ZM 90, BDK 91]. Unfortunately, there has been less progress on understanding the principles of OODBs. This is in marked contrast with the elements of relational database theory, see [U 88] and [vL 90–ch.17]. Generally accepted definitions of object-oriented data models (as in [A+ 89]) are still no more than a list of desirable concepts, with little integration or analysis. The concepts themselves can be divided in two categories, which reflect their origins: from relational database theory or from object-oriented programming. Each of these *two facets of object-oriented data models* involves both data description and data manipulation. So the division is not one of type-description vs type-manipulation, but rather one of concrete (or *structural*) vs abstract (or *behavioural*) type disciplines. Key issues are formalizing the individual facets and their integration.

In this short survey we: (1) present succinct, but still fairly detailed, formalizations for the two facets, and (2) summarize their analysis from [AK 89], for the structural, and [AKW 90], for the behavioural part. For each the format is an example-driven, two page outline. We close with examples [A 89, LR 89, BDK 91] integrating the two facets, a description of some promising research directions and a selected (but incomplete) list of references.

1. The Structural Data Model: The data description generalizes existing "nested relation" and "complex structure" data models, in particular [AB 87, KV 84]. Our data manipulation language IQL generalizes existing rule-based, statically typed database query languages, and can be used to compare their expressive power. After some preliminary notation, we define database schemas and their instances, almost as succinctly as for the relational data model! Assume the following countably infinite and pairwise disjoint sets: (1) *relation names* $\{R, R', \ldots,\}$, (2) *class names* $\{C, C', \ldots,\}$, (3) *attributes* $\{A, A', \ldots,\}$, (4) *base constants* $B = \{b, b', \ldots,\}$, and (5) *object identifiers* or *oids* $O = \{o, o', \ldots,\}$. The set of *o-values* is the smallest set containing B and O and closed under finite tupling ($[A_1 : v_1, \ldots, A_k : v_k]$ for distinct A's) and finite subseting ($\{v_1, \ldots, v_k\}$). An *o-value assignment* ρ maps relation names $\mathbf{R}$ to finite sets of o-values. A [*disjoint*] *oid assignment* π maps class names $\mathbf{C}$ to [pairwise disjoint] finite sets of oids. A *value map* of an oid assignment π is a *partial function* ν associating the oids in π to o-values. Intuitively, o-value assignments are like "relational database instances", oid assignments put "objects" into "classes", and value maps associate "objects" to "values" or when undefined to "null values". A fine point is the cyclic use of class names $\mathbf{C}$ and oid assignment π, respectively, in the syntax and semantics of types as follows: (1) Type expressions *types*($\mathbf{C}$) are defined by the grammar $\tau = \mathbf{C} \mid \emptyset \mid B \mid [A_1 : \tau, \ldots, A_k : \tau] \mid \{\tau\} \mid (\tau \vee \tau) \mid (\tau \wedge \tau)$. (2) The matching type domains are the sets $Dom = \pi(\mathbf{C}) \mid \{\} \mid \{b, b', \ldots\} \mid [A_1 : Dom, \ldots, A_k : Dom] \mid \{Dom\} \mid (Dom \cup Dom) \mid (Dom \cap Dom)$.

Definition DB: A *database schema* consists of a finite set of relation names $\mathbf{R}$, a finite set of class names $\mathbf{C}$ and a function T from $\mathbf{R} \cup \mathbf{C}$ to *types*($\mathbf{C}$). A *database instance* of such a schema consists of an o-value assignment ρ to $\mathbf{R}$, a disjoint oid assignment π to $\mathbf{C}$, and a value map ν of π such that:
(1) each R in $\mathbf{R}$ contains o-values of the right type, $\rho(R) \subseteq Dom(T(R))$,
(2) each C in $\mathbf{C}$ contains oids mapped to o-values of the right type, $\nu(\pi(C)) \subseteq Dom(T(C))$, and
for o [not] a set-valued oid, undefined $\nu(o)$ is $\nu(o) = \emptyset$ [undefined $\nu(o)$ is a null value]. □

*INRIA, Rocquencourt, FRANCE. (abitebou@inria.inria.fr). Supported by the Projet de Recherche Coordonnée BD3.

†Brown Univ., Providence RI, USA. (pck@cs.brown.edu). Supported by NSF grant IRI-8617344 and ONR grant N00014-83-K-0146 ARPA Order No. 6320-1. Also, would like to thank Sridhar Ramaswamy for his help with the presentation.

The types provided in this structural data model are concrete, as opposed to abstract. They include records, sets (lists could be handled similarly) and pointers. In addition, non-disjoint oid assignments and intersection and union types can be used for expressing *structural polymorphism* and in particular *structural inheritance*; because of limited space we refer to [AK 89] for details on this issue. Also in [AK 89], we present a corresponding *pure value* data model without oids, based on the regular infinite trees produced by iterated applications of the value map ν. We now present an example that illustrates the data model's substantial descriptive power (from Genesis no less!).

Example DB: Consider a schema with two class names *C-1st-generation*, *C-2nd-generation* and with two relation names *R-founded-lineage*, *R-ancestor-of-celebrity*. Their types refer to a base domain, e.g., *B-string*, and to class names, e.g., *C-1st-generation*, but not to relation names. Union is allowed. Then consider an instance of this schema, whose oid's are *o-adam, o-eve, o-cain, o-abel, o-seth, o-nameless*. Note the cycles.
T(*C-1st-generation*) = [name: *B-string*, spouse: *C-1st-generation*, children: {*C-2nd-generation*}]
T(*C-2nd-generation*) = [name: *B-string*, occupations: {*B-string*}]
T(*R-founded-lineage*) = *C-2nd-generation*
T(*R-ancestor-of-celebrity*) = [anc: *C-2nd-generation*, desc: (*B-string* $\vee$ [spouse: *B-string*])]

π(*C-1st-generation*) = { *o-adam, o-eve* } and π(*C-2nd-generation*) = { *o-cain, o-abel, o-seth, o-nameless* },
ρ(*R-founded-lineage*) = { *o-cain, o-seth, o-nameless* },
ρ(*R-ancestor-of-celebrity*) = { [anc: *o-seth*, desc: Noah], [anc: *o-cain*, desc: [spouse: Ada]] },
ν(*o-adam*) = [name: Adam, spouse: *o-eve*, children: { *o-cain, o-abel, o-seth, o-nameless* }],
ν(*o-eve*) = [name: Eve, spouse: *o-adam*, children: { *o-cain, o-abel, o-seth, o-nameless* }]
ν(*o-cain*) = [name: Cain, occupations: { Farmer, Nomad, Artisan }],
ν(*o-abel*) = [name: Abel, occupations: { Shepherd }],
ν(*o-seth*) = [name: Seth, occupations: {}], ν(*o-nameless*) is a null value. □

The structural data model comes with a "complete" query language. This language, IQL, is Datalog with negation combined with set/tuple types, invention of new oid's, and a weak form of assignment. With no additional syntax, inflationary negation can express sequential execution and while-loops. IQL was designed as a minimal rule-based formalism for expressing all computable queries. This precise expressive power was achieved – modulo a technical condition (copy elimination). The design was influenced by both the COL language of [AG 88], for the manipulation of sets, and the detDL language of [AV 88], for the invention of new oid's. The following example illustrates all of its important features on an efficiently executable query. This query is not expressible in most other database languages, but is easily expressible in any programming language (e.g., Pascal).

Example IQL Query: Our type system 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 cyclic. 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. More formally, let the input schema be just a relation R with $T(R) = [A_1{:}B, A_2{:}B]$ and the output schema be a class C with $T(C) = [A_1{:}B, A_2{:}\{C\}]$. The input instance I represents directed graph G over B nodes. The desired query is to transform the "flat" I into a "deep" output instance J that also represents G (where now the nodes are "objects"). Let us examine the computation in IQL in four separate stages. Using simple techniques from [AV 88], one can modify the rules (adding inflationary negation) to force the stages' sequential execution.

$R_0(x) \longleftarrow R(x, y)$ and $R_0(x) \longleftarrow R(y, x)$. In this first stage, we produce (in standard Datalog fashion) the set of node names. We use a relation R_0 with $T(R_0) = [A_1 : B]$.

$R'(x, p, p') \longleftarrow R_0(x)$. In this second stage, we invent two oid's per node, using the semantics of [AV 88]. We use a relation R' whose tuples contain oid's from class C and from a temporary class C', that is, we have the types $T(R') = [A_1 : B, A_2 : C, A_3 : C']$ and $T(C') = \{C\}$. This stage's rule invents two oid's for each node, one of which will go into class C and the other into class C'. Note how the variables p, p' in the head are not in the body. When the new oid's o, o' are invented by instantiating p, p' they are placed in the proper classes and they are automatically assigned default values: $\nu(o)$ is undefined and $\nu(o')$ is the empty set, because of the set valued type of C'.

$\widehat{p'}(q) \longleftarrow R'(x,p,p'), R'(y,q,q'), R(x,y)$. In this third stage, we nest the oid's representing nodes in C into sets of successors of a node. Here p' is set valued and its value, noted $\widehat{p'}$, is a set in which the corresponding q's are collected. The nesting is done by using the oid's of C' as temporary names that simulate [AG 88]'s data-functions.

$(\hat{p} := [x, \widehat{p'}]) \longleftarrow R'(x,p,p')$. In this final stage, the nodes of C have been grouped into C', and the connection in R' between x, p, p' is used to produce the desired result. This weak form of assignment is performed only when $\hat{p}$, the value of p, was undefined. It is a single-assignment-form, i.e., no further changes are made to $\hat{p}$. □

Analysis of the Structural Data Model: Our main contributions in [AK 89] are the succinct description of the data model, and the design/analysis of IQL. This query language can be statically type checked, can be evaluated bottom-up and naturally generalizes many rule-based languages. Interestingly, IQL 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 – up to copy elimination. This last property is a "completeness" theorem – modulo copy elimination (which is not expressible in IQL). However, IQL can express *all* computable queries on pure values. □

2. The Behavioural Data Model: We propose method schemas as a core data model of the object-oriented programs used in most OODB prototypes. We believe that our functional formalism is the most natural one and that, independent of formalism, the cases stressed in [AKW 90] (small-arity, recursion free method schemas) should be central to any object-oriented data model. The formalism is program schemas [vL 90–ch.9], that express *composition*, *recursion* and *if-then-else*, and that respect *encapsulation*; the manipulation of objects only via methods. We do not use the lambda calculus [vL 90–ch.7], because most OODB prototypes do not have higher order functions.

A *method schema* is an *isa* forest (or single inheritance) hierarchy of *classes*, that is associated with *method* (or program) definitions, see Figures 1–3. Each *object*, in a database instance of a method schema, is created in a single class where it *belongs*. All objects belonging to the same class have the same methods, so methods are part of the schema. Objects have methods explicitly defined at their class or implicitly inherited from the ancestor classes of their class in the *isa* hierarchy (see syntax below). Programs are interpreted operationally using graph rewriting (see semantics below). Each object has no other visible structure, so we have algebraic specifications [vL 90–ch.13].

Method Syntax: There are two kinds of explicit method definitions at class names $c_1, ..., c_n$, $n \geq 0$, where at most one explicit definition is allowed at each $c_1, ..., c_n$. A *base* method definition $m@c_1, ..., c_n$ is a finite function (of arity $n \geq 0$) which has name m and has a signature $c_1, ..., c_n \rightarrow c_{n+1}$. A *coded* method definition $m@c_1, ..., c_n$ has name m and is associated with an n-term, that is, a finite rooted directed acyclic graph whose nodes are labeled by method names and class names $c_1, ..., c_n$ (at input nodes $1, \ldots, n$) and whose arcs are ordered at each node. (The idea is that n-terms represent functions, built from the base functions via composition and recursion; method inheritance will provide if-then-else). Three 1-terms t, t', t'' are shown in Figure 2, where arcs are ordered left-to-right. Figure 3 contains an explicit method definition example. In addition to explicit definitions, methods of a class can be implicitly inherited by its descendants. (The idea is to have a restricted form of *code polymorphism* for the convenience of code reuse). For flexibility, we allow reusing method names in different parts of the *isa* hierarchy, however, we keep the sets of base and coded method names distinct. Inheritance means that we can have at most one explicit and possibly several implicit definitions for a method name at the same class names, i.e., it implies *method name overloading*. Resolution of method name overloading consists of finding, for a given method name and given classes, a unique definition. We use the closest ancestor in the *isa* hierarchy resolution rule. For the multi-argument case we take the argument-wise closest ancestor, see [AKW 90]. For example, the method *cost* in Figures 1–3 is explicitly defined on *part*, implicitly inherited by *basepart*, *comp_part*, and explicitly redefined on *basepart* (the overloading here is resolved in favor of the explicit definition); it is also explicitly defined for *pair_of_parts*. □

Method Semantics: For the base methods each database instance contains finite interpretations that respect the given signatures. That is, $m@c_1, ..., c_n \rightarrow c_{n+1}$ is materialized as a finite function which for any tuple of objects belonging to $c_1, ..., c_n$ gives an object belonging to c_{n+1} *or to one of its descendant classes*. The interpretations of coded methods are defined recursively using graph rewriting, just like program schemas. However, because of name overloading, a given method name in a term is interpreted based on its context, i.e., on the classes of its arguments.

(The idea is to model *late binding*). We now give some intuition for the rewriting of coded methods. Consider a depth-one n-term, whose one internal node is labeled m and whose inputs have been replaced by objects, as a procedure call to m. Based on the classes of the arguments, we can replace (or reduce or graph rewrite) this call either by some object/code if it is defined in a base/coded fashion, or by an error message if it is undefined. In general, given the n-term with objects substituted for class name inputs, in order to compute we reduce the first (in the term's depth first ordering) method name with instantiated leaves as children. If this deterministic reduction process terminates after a finite sequence of graph rewritings then we will either (1) obtain a result (i.e., an object) or (2) reach an inconsistency (i.e., get an error message). A partial sequence of graph rewritings for the method *cost* in Figure 3 is shown in Figure 4, where o is in class *pair_of_parts* and o', o'' are in class *basepart*. □

In [AKW 90] we analyze the nontrivial problem of *method schema consistency*. More precisely, we want to check whether a given method schema can produce an inconsistency, for some finite interpretation of the base methods as part of some finite terminating computation. *The Problem of Type Inference in OODBs* can be modeled by the static method schema consistency question. Our analysis of this reachability problem primarily produces signatures for the coded methods. *The Problem of Managing Schema Evolution in an OODB* can be modeled by the incremental method schema consistency question. This is an example of dynamic type inference. In general, the static method schema consistency problem is undecidable. The difficulties come from the simultaneous presence of recursion and the use of multi-argument methods to represent contexts. As in program schemas linear recursion and two arguments suffice for undecidability. Practical programs are often less complex. Practical method schemas involve mostly the one-argument (monadic) and/or the recursion-free cases. Also, signatures are *covariant* or *contravariant*.

Analysis of the Behavioural Data Model: In [AKW 90] we show that in the monadic and recursion-free case, consistency checking can be done using finite state automata in PTIME, just like lexical analysis in compilers. In this case, inheritance and name overloading introduce nondeterminism but covariance removes it. In favor of covariance, we give DLOGSPACE vs NLOGSPACE arguments and linear vs quadratic time algorithm arguments. In the monadic recursive case we also have decidability, using context free language emptiness. In the multi-argument recursion-free case decidability follows from an exhaustive search, at the expense of co-NP-completeness even for arity two. Interestingly, in the recursion-free covariant case there is a PTIME test for a single two-argument coded method. We use our case analysis as the basis for a general heuristic for the static consistency problems. Enhancements (such as varieties of multiple-inheritance, more precise or less precise signatures, attachment of methods to first argument, and virtual classes with no objects belonging to them) can be integrated and studied with this core data model. □

3. Combining the Two Facets: Despite their different nature, it is possible to combine the two facets in one data model. Two examples, familiar to the authors, are the O_2 object-oriented prototype (V1 version) [BDK 91] and the extension of IQL with abstract data types [A 89]. The integration of the two styles of data description is relatively simple. The more complicated task is the integration of data manipulation; for imperative formalisms in O_2 see [LR 89] and for logical formalisms see [A 89]. One technical problem, for OODBs, is the interaction of structural and behavioural inheritance with type checking and type inference. In [AK 89], we propose a compilation of structural inheritance into union types, that preserves static type checking and the "completeness" of IQL. However, this static solution does not capture the common dynamic use of inheritance in object-oriented programming languages (as is done in [LR 89] at the expense of full static type checking). The reason is that union types are hard to implement. Is it possible to have dynamic use of inheritance in the query language, without giving-up static type checking or "completeness"? Another technical problem, important for OODBs, is managing schema evolution. This is motivation for the analysis of incremental method schema consistency, with structural features.

4. More Expressive and Efficient Types: We close our exposition with a description of two additional problem areas. (1) How does one add *polymorphism and higher order functions* to the type system? Inheritance provides only limited polymorphism. There has been a great deal of research on ML with records and inheritance. Can one use such a programming language for an object-oriented data model? Can one take advantage of the various polymorphic higher order calculi developed in recent years, surveyed in [vL 90–ch.8]? (2) The most important question, concerns efficient implementations. The structural data model inherits much of the existing database optimization technology [U 88], but *query optimization* remains open for the behavioural (and the combined) data model(s).

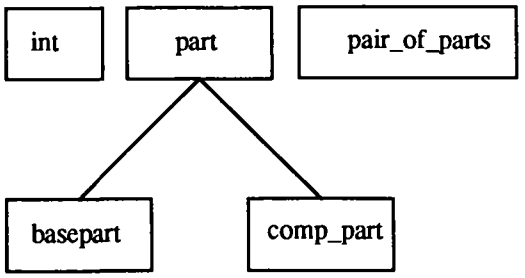


Figure 1: *isa* hierarchy

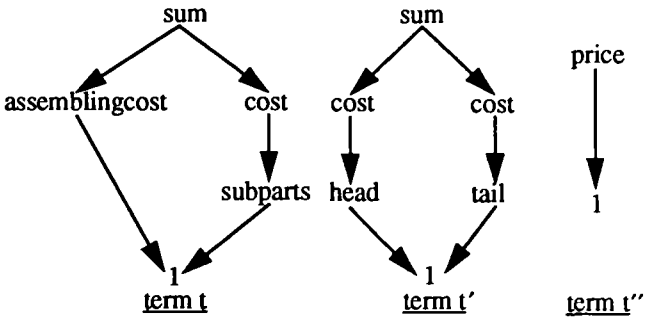


Figure 2: terms

method sum	@int,int	: int	method subparts	@part	: pair_of_parts
method head	@pair_of_parts	: part	method assemblingcost	@part	: int
method tail	@pair_of_parts	: part	method cost	@basepart	= t''
method price	@basepart	: int	method cost	@pair_of_parts	= t'
			method cost	@part	= t

Figure 3: Methods

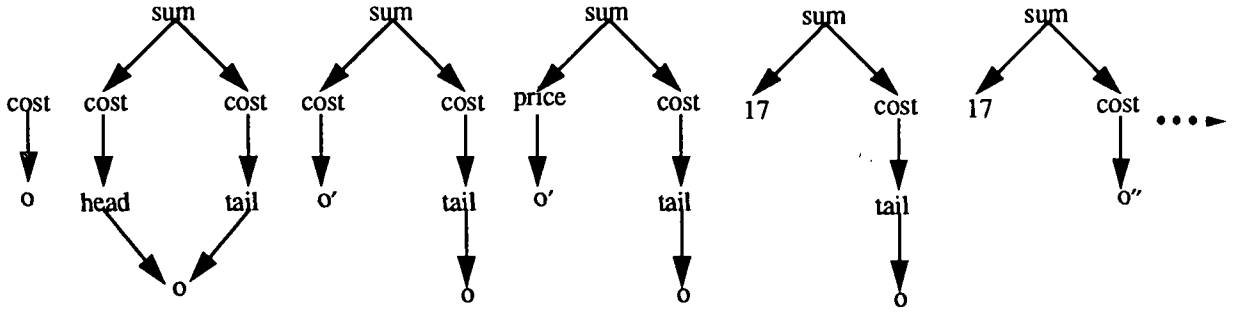


Figure 4: A (partial) sequence of rewritings

5. References

[A 89] S. Abiteboul. "Towards a Deductive Object-Oriented Database Language". 1rst DOOD, 419–438, 1989.

[AB 87] S. Abiteboul, C. Beeri. "On the Power of Languages for the Manipulation of Complex Objects". INRIA-846, 1988.

[AG 88] S. Abiteboul, S. Grumbach. "COL: a Logic-based Language for Complex Objects". 1rst EDBT, 271–293, 1988.

[AK 89] S. Abiteboul, P.C. Kanellakis. "Object Identity as a Query Language Primitive". SIGMOD, 159–173, 1989. Also, INRIA-1022, 1989.

[AKW 90] S. Abiteboul, P.C. Kanellakis, E. Waller. "Method Schemas". 9th PODS, 16–27, 1990.

[AV 88] S. Abiteboul, V. Vianu. "Procedural and Declarative Database Update Languages". 7th PODS, 240–250, 1988.

[A+ 90] M. Atkinson, etal. "The Object-Oriented Database System Manifesto". 1st DOOD, 40–57, 1989.

[BDK 91] F. Bancilhon, C. Delobel, P.C. Kanellakis (eds). "Building an Object-Oriented Database System: the Story of O_2". Morgan Kaufmann, to appear 1991.

[KL 89] W. Kim, F.H. Lochovsky (eds). "Object-Oriented Concepts, Databases, and Applications". ACM Press, 1989.

[KV 84] G. Kuper, M.Y. Vardi. "The Logical Data Model: a New Approach to Database Logic". 3rd PODS, 86–96, 1984.

[LR 89] C. Lecluse, P. Richard. "The O_2 Database Programming Language". 15th VLDB, 411–422, 1989.

[U 88] J.D. Ullman. "Database and Knowledge-Base Systems: Volumes I and II". Computer Science Press, 1988.

[vL 90] J. van Leeuwen (ed). "Handbook of Theoretical Computer Science: Volume B". North Holland, 1990.

[ZM 90] S.B. Zdonik, D. Maier (eds). "Readings in Object-Oriented Database Systems". Morgan Kaufmann, 1990.

Theoretical Foundations for OODB's — a Personal Perspective *

Catriel Beeri
Department of Computer Science
The Hebrew University of Jerusalem

1 Introduction

Object-orientation is a major paradigm in current database research. An enormous amount of system work is invested in development of object-oriented databases (OODB's); but all seem to agree that the theoretical foundations can best be described as fuzzy. For me, 'theory' or 'formal foundations' means *logic* (not necessarily predicate calculus). There is much to be said for the logical approach to OODB theory, but for brevity I assume that the reader is already convinced. Let me only emphasize that I am looking only for logics that are axiomatizable, for obvious reasons. The following is a short account of some directions that I believe are relevant to the development of a logical foundations for OODB.

The OODB paradigm is rich in ideas and concepts. I do not believe that a comprehensive theoretical foundation for it can be developed easily, in one big step. My approach has been to take as a starting point a familiar and well understood theory, and try to develop extensions that each covers a useful and interesting category of OODB concepts. Hopefully, such extensions can be merged into progressively more general theories. I try to argue below that, indeed, there are several directions to go, that each can be used to provide a foundation for some interesting features of OODB's, and that they can be combined with each other. In the given space, it is impossible to provide a detailed account. I discuss one (simple) direction rather extensively, and others only fleetingly. (But be warned: the really interesting results are to be found in those other directions). Note: The ideas below are not all originally mine; some have been discussed extensively in the literature. I have not included a bibliography, but some pointers are provided.

2 The Initial Model Approach

During the 80's we have witnessed the appearance of models such as *nested relations* and *complex objects*. Add to these *object identity*, the ability to define and use ADT's, and finally the often expressed requirement that the database and the application programming language should have compatible type structures. In short, a primary characteristic of OODB's is a powerful and extensible data type facility. Data types belong to the programming language domain. What can we

*Research partially supported by a grant from the USA-ISRAEL Binational Science Foundation.

borrow there? Let us consider the theory of ADT specifications.[1]

Recall that our viewpoint about relational databases changed about ten years ago from model theoretic to proof theoretic: a database is a set of (ground atomic) formulas. The semantics of such a database was delineated in works by Reiter, that introduced additional, implicit, axioms, and defined the meaning of a database as the logical closure of the explicit and the implicit axioms. The implicit axioms state that the constants in the database are the only elements (*domain closure*), that they are distinct from each other (*unique naming*), and lastly, the well known CWA: relationships not given explicitly in the database are false. There is also an equivalent model-theoretic semantics since, given these axioms, the model is unique. The theory of deductive programming is a generalization: the domain is obtained from the constants given in a program by closure under (syntactic) function application — this is the Herbrand universe. The elements of this universe are assumed to be distinct. Obviously, this is a generalization of the domain closure and unique naming assumptions. Finally, the semantics of a program is the set of ground atomic formulas it implies, or equivalently the *minimal model* in the Herbrand universe — a generalization of the CWA.

Now, consider how abstract data types are specified. The approach is logic-based: The language contains function symbols (including constants), and the only predicate symbol is equality. Thus, the only atomic formulas are equations. Specifications are presented as sets of (universally quantified) equations or conditional equations, that is, Horn-clauses. The models are *algebras*. The most common definition of the semantics of a specification is its *initial algebra* or model.[2] Initial models are unique, up to isomorphism, so this definition is indeed *abstract* — it defines the semantics uniquely only up to isomorphism, hiding representation details.

A well known construction of an initial model is the following: Start with the *term algebra*, whose domain is the Herbrand universe — the collection of all terms, and where the 'application' of a function f to terms $t_1, \ldots, t_n$ is defined to be the term $f(t_1, \ldots, t_n)$. The extension of the equality predicate defined by (conditional) equations is a congruence relation on the terms. The quotient of the term algebra by this relation is also an algebra, and it is an initial model. (We take the quotient so that equality behaves 'normally', that is, two elements are 'equal' iff they are the same element.) For some intuition, consider the definition of stacks, with operations $push, pop, top, empty$, where the equations include, e.g., $pop(push(n, s)) = s$. The 'stack' elements of the term algebra are those obtained from *empty* by applications of *push* and *pop*; but the equations make elements like s and $pop(push(n, s))$ equivalent. The quotient algebra is precisely what we intuitively expect.

An important property of the initial model is that elements denoted by different ground terms are the same, that is a ground equation holds in it, precisely when the equation is logically implied by the specification. This reflects a proof-theoretic interpretation of the semantics, as the logical closure of the specification. Note the similarity to the semantics of deductive programming. Can then the two approaches be unified? Yes, easily! Take a language with both function and predicate symbols. For any set of Horn clauses, there exists a (unique up to isomorphism) initial model. Its elements are the equivalence classes of terms modulu the congruence relation defined by the

[1] The ideas below have been discussed in several papers by Goguen et al; see, e.g, in D. DeGroot, G. Lindstrom (eds.) *Logic Programming: Functions, relations, and Equations*, Prentice Hall, 1986.

[2] Initiality is easy to define, but we cannot go into details for lack of space.

formulas for the equality predicate and, in particular, the domain is the Herbrand universe iff there are no nontrivial equations in the logical closure. It is still the case that a ground atomic formula holds in it iff it is logically implied from the given program. Thus, the logic programming paradigm is obtained as a special case, where the equality predicate is not used. Relational databases are obtained by also dropping the function symbols, and allowing only atomic ground formulas. All the theorems that underlie deductive programming hold in this more general framework. In short, it unifies the theories of relational databases, deductive programming, and ADT's.

A unification of several theories is in itself a gratifying achievement. But, there are also immediate benefits: We can now (try to) generalize techniques and results of the more limited theories; this often succeeds, and even when it fails it provides insight. We can relate results that were obtained independently in seemingly distinct domains, and often reinterpret them as special cases. We can also investigate issues that exist only in the generalized framework. All this, of course, is just advertisement; let me mention some concrete examples.

The initial model approach allows one to define a variety of generic data types, in particular tuples, finite sets, lists, and so on. Thus, this framework includes nested relations/complex objects models, in which the values are atoms, tuples, and sets. It also allows user defined ADT's. For example, to define finite sets, we use the constant *empty*, and the function *insert*, that takes an element and a set, and outputs a new set obtained by inserting the element into the given set. The equations state that the order of insertion is irrelevant, and inserting an element twice has the same effect as inserting it once. Membership, union, and so on can similarly be defined.

Why should we care to formalize a simple concept such as sets? A restricted formalization of sets is embedded in the notion of *model* of the predicate calculus, and is used in relational databases: Certain sets (of tuples) are given, namely the predicate extensions, other sets are defined from them by formulas; one can ask if an element is in a set, or print all elements of a set. But, in the new models sets are values that can be manipulated, taken apart and reconstructed. We must tell the system how we understand sets. Trying to manipulate set naively may be dangerous: In unrestricted LDL one can express the paradoxes of set theory. A formalization, as above, helps us to discover what can be done with sets safely and efficiently. As an example, the existence of a least fixpoint semantics for LPS, proved by Kuper using specialized techniques, follows as a corollary of the general theorems. Features of languages that allow sets as elements such as LPS, LDL, COL can now be compared and explained These are the benefits of having a general theory.

In this approach we cannot model updates, hence we cannot model the fact that object identity does not change even when its value is updated. But, a simple static notion of object identity can be modeled by a specification that defines pairs where one component, the identity, is taken from some abstract domain (where the only operation is comparison), and the other component is the value.[3] For query languages, this is sufficient.

Traditionally, there have been two approaches to query languages: calculus-based and algebraic. Now, we can view a database in our framework in two ways: First, as a set of formulas involving, say, predicates $p_1, \ldots, p_n$; second, as a collection of named sets $s_1, \ldots, s_n$ (whose contents is defined

[3]This may be rather disappointing, but without modeling updates, we cannot hope for much more.

by equations). In the first approach, a query is obtained by augmenting the database by additional formulas, then asking what are the elements t such that $p(t)$ holds, for some p. This is the calculus-based approach: a formula defines an unnamed predicate, which is the query predicate. Deductive languages also fall in this category. In the second approach, we apply functions to the given sets, to produce new values. Appropriate generalizations of all the relational algebra operations are definable as functions in this framework[4], and so are other useful functions such as aggregates. This sheds new light on the algebra vs. calculus dychotomy: these are functional and predicative programming styles, respectively. Are they equivalent in this general setting? The answer is positive, for both the nonrecursive and the recursive cases, under certain conditions. this generalizes many previous results, and also uncovers some limitations.

The theory of ADT specification is usually couched in terms of *many-sorted* logic, where the algebras, or models, have distinct domains for the sorts. But, actually we have here two independent ideas: that equality can be used to define (state properties of) data types, and that typing can be incorporated into the calculus, allowing functions that are meaningful only on certain subsets of the domain. These ideas can be used independently or together. [5] The many-sorted approach has the benefit of simplicity. For example, one may model subtyping (a form of *isa* relationship) by mappings between domains in the initial model, and investigate its properties.[6] Of course, ultimately we would like to generalize the framework to more sophisticated type disciplines.

3 Higher-Order Concepts

Some relational systems now store queries; all OODB's store functions in the form of methods. Various features of the behavioral aspects of OODB's, such as inheritance and overiding, are neatly modeled when new functions can be defined from given functions (see my paper in DOOD89). The requirement of having a complete database programming language also entails having a powerful function definition facility. In short, we want functions as first class values, and a flexible function definition facility.

The formalism underlying functional languages is the (untyped or typed) λ–calculus. This is also a logic, but it does not contain connectives and quantifiers, nor predicates. In the 30's, Church tried to extend λ–calculus to a full logic; his system was shown to be inconsistent. Nevertheless, he later presented a consistent logic that combines the predicate calculus and the typed λ–calculus.[7] I believe that the idea also works for the untyped λ–calculus.[8] An important benefit of this logic is that λ-abstraction can be applied to predicates, not just to functions. This further increases its power. This logic is a considerable generalization of the framework of the previous section, where the first-order functional language with equality is replaced by the λ–calculus. A database

[4] It is well known how to define a function *map* that given a function f and a list, outputs the list with f applied to each element. There is no difficulty to adapt thdefinition to obtain such as *select* and *project* on sets.

[5] We can define, say, integers and stacks in a mono-typed logic, but then *we* have to careful never to write an expression that applies + to a stack. Typing is a natural solution.

[6] See my paper with T. Milo in PODS 91.

[7] See the paper by Nadathur and Miller in JACM, Oct. 90.

[8] However, in view of Church's experience, do not believe until you see a proof in print.

is described, as before, by a collection of Horn clauses. If we carry over the rest of the paradigm, that is the semantics as the logical closure, we have a very attractive framework, combining the full powers of relational and functional programming, that can be used with various expressive type systems.

An important characteristic of OODB's is the ability to treat *higher-order*, or schema-level, entities as first class values. Functions are just one example. Others include sets, attributes, relations, classes and types. For example, *isa* relationships relate such entities, and we would like to reason or query about such relationships. The issue of a logic where higher-order elements can be manipulated as if they were first-order has been investigated by Kifer and coauthors; see his paper in this journal. Let me consider just one issue. We have seen above how to models sets as values. There is another approach to sets. If we make a selection from a binary relation on an element that appears in the first position, we can view the result as representing a set associated with that element, and the element as an object id. Thus, we can in this way model (some aspects of) set-valued objects, which we may call *classes*. We can now model relationships between classes; e.g., *isa* is a binary predicate with axioms that make it a partial order.[9] These are two of the 'secrets' of F-logic. We can similarly model binary-relation valued objects, and with axioms that make the relations functional, we have function-valued objects. Interestingly, the λ–calculus has an axiomatization and 'first-order' models, and the constructions of these models utilizes precisely this idea of associating functions with elements, with certain closure properties that ensure that λ abstraction is always defined. Thus, there are common ideas underlying the various 'feasible' higher-order logics. It is important to note that the idea above is important for the designer of the logic, since he needs effective proof procedures. A language should contain enough sugar to let the users see just sets, functions, classes and so on. Users need not be bothered about how the proof or model theory were developed.

4 Conclusion

I have shown that existing logic-based approaches to databases and programming languages promise to be useful for modeling OODB's. I hope that this extremely sketchy outline has convinced you that the approach is interesting and potentially useful. In particular, that there is a small set of concepts underlying the various higher-order logics. Often they are independent, and can be combined to suit our needs. By elucidating them, we can hope to obtain a toolkit for constructing expressive logics to suit our needs.

Much remains to be done in constructing new theories and extending known results, and in providing satisfactory modeling of OODB's. For the theoretically inclined, there is a wealth of problems waiting to be investigated. I see the development of a full scale foundation for OODB's as an endeavor that may require several years. This is no reason to be alarmed; this is how science progresses. I see a good reason to be excited and interested about this research; for if the approach I have described works, then the results will be a unified foundation for two of the central fields of Computer Science — databases and programming languages.

[9] The tricky part is, however, how to model inheritance. See F-logic.

A First-Order Formalization of Object-Oriented Languages *

Michael Kifer
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
kifer@cs.sunysb.edu

1 Introduction

The last few years saw a number of attempts to provide a formal foundation to the object-oriented programming paradigm. In a nutshell, one of the most salient features of this paradigm is the use of data abstraction, i.e., encapsulated objects with complex internal structure accessible through publicly known functions, called *methods*. Semantically similar objects are classified into *classes* and the latter are organized in a hierarchy, sometimes called *IS-A* hierarchy.

IS-A hierarchies are useful because they help factor out information that is shared by most of the members of a class. Such information is explicitly stated for the class and then is implicitly *inherited* by subclasses and individual objects. Subclasses and objects that do not share the common information constitute *exceptions* and can overwrite the inheritance. This kind of inheritance is called *non-monotonic* and has been extensively studied in the literature.

Another important idea underlying object-oriented languages is the notion of a *type*. Generally, the term *type* refers to an arbitrary set of abstract values. The collection of all types used in the language is called a *type system* and is specified via a *type expression sublanguage*. In object-oriented languages, types are used to specify collections of structurally similar objects, where the "similarity" is manifested by a common set of methods applicable to the objects of the same type. Methods are also typed, i.e., they are declared so as to expect arguments of certain types and return objects of certain types.

Since objects populating a class share *semantic similarity*,[1] it is reasonable to expect that they would be *structurally* similar as well. Moreover, the type of objects in a class C is inherited by every subclass of C and so the type of a subclass is a subset of the type of a superclass. We therefore believe that in object-oriented languages typing is secondary to the semantic classification into classes. This also explains why in many languages classes and types are closely, yet mysteriously, related.

In this short paper, we shall provide an elementary introduction into the results of our recent work on the logical foundations of object-oriented and frame-based languages. Our goal is to clarify the central concepts of such languages and outline a simple logical theory for them. We start with a very basic object-centered language, roughly corresponding to Maier's O-logic [8] (but, in fact, dating back to the mid-seventies to the little-known works of Pawlak [9, 10]), and proceed by adding more features, going on the way past what is

*Work supported in part by the NSF grant IRI-8903507

[1] Classes that represent heterogeneous collections of objects do not contradict this claim, for—from the data modeling point of view—semantic affinity is the only proper reason for placing different objects in the same class.

known as C-logic [1], the revised O-logic [5, 6], and eventually accounting for many of the features of F-logic [2, 3].

2 Object-Centered Languages

By an object-centered language we mean one that supports (to various degree) the concept of an object with a complex internal structure. We start with what we call a "simple O-logic" and continue to add features until we (almost) reach the functionality of the full O-logic, as described in [6].

Central to all object-centered languages is the notion of *object identity*—a simple concept that generated heated debates in the database community. Our view is that objects are abstract or concrete entities in the real world and object identity is a syntactic wherewithal needed to refer to these objects in the programming language. More accurately, object identity is a purely implementational notion, a surrogate or a pointer to an object, while the syntactic counterpart in the programming language is a *denotation* of an object.

For the reason that is not completely clear to me, some recent proposals tried to incorporate the notion of object identity without introducing object denotations explicitly into the language. This was essentially the reason why Maier's O-logic [8] was unsuccessful in dealing with databases that contain anything more complex than a simple collection of objects. LDM of Kuper and Vardi [7] was more lucky in that respect, but only at the price of severely limiting the language and the queries.

In our model, objects are referred to via their denotations, which are nothing else but first-order terms, such as *13, _256, john32.* Function symbols can be used to construct more complex denotations, such as *father(mary), head(csdept(stonybrook)).*

2.1 Simple O-logic—The Dual of the Relational Model

The central concept of the relational model, the relation, is defined as a set of tuples, where a tuple is a function from the set of attributes of the relation into the domain.[2] The dual concept can be stated as follows:

> *An* attribute *is a function that maps the set of objects into the domain.*

This dual data model is actually rather old; it appeared in a little-known paper by Pawlak [9] and was subsequently used in [10] and in some of the later papers of Lipski.[3]

It is interesting to note here that Pawlak's model is *not value-oriented*, i.e., it is perfectly legal that all attributes will map two distinct objects into the same values. In contrast, the relational model is value-oriented: it is impossible for two different tuples to map attributes in the same way. In other words, if $\mathcal{A}$ denotes the set of all attributes then in Pawlak's model $\forall a \in \mathcal{A}(\, a(obj_1) = a(obj_2)\,)\ \&\, obj_1 \neq obj_2$ is possible, while in the relational model $\forall a \in \mathcal{A}(t_1(a) = t_2(a))\ \&\, t_1 \neq t_2$ is impossible.

Being cast in the concrete syntax used by Maier in [8] and incorporating the aforesaid idea about object identity, Pawlak's model yields what we call *Simple O-logic*—a logical, object-centered language that Maier would have arrived at, should he overcome the problems encountered on the way (which, in my view,

[2] Assuming, for simplicity, that all attributes share the same domain.

[3] Ironically, these works were published long before the beginning of the recent craze with object-oriented languages and so Pawlak did not get due credit. This also applies to LDM of Kuper and Vardi whose work might have attracted more attention if they only used the "right" buzz-words.

stemmed from an attempt to emulate ad hoc implementational ideas too closely). Simple O-logic allows for the representation of facts such as

$$john[name \rightarrow \text{"}John\ Doe\text{"};\ salary \rightarrow 20000]$$
$$father(mary)[address \rightarrow \text{"}MainSt.\ USA\text{"};\ spouse \rightarrow sally]$$

where *john*, *father(mary)*, and *sally* are denotations of objects that are intended to represent information about persons, *"John Doe"*, *"Main St. USA"* are denotations of string-objects, and 20000 is a denotation of an integer-object; *name*, *salary*, *address*, and *spouse*, are attributes. Besides the basic facts about objects, one can write deductive rules, just as in regular deductive databases. Details can be found in [5, 6].

2.2 O-logic Revisited

Simple O-logic accounts only for a very limited type of complex objects. For instance, there is no straightforward way of saying that John has children, Mary, Bob, and Alice. Nevertheless, it is very easy to extend this logic to support full-fledged complex objects. We just have to allow some attributes to be *set-valued*, i.e., be defined as functions from objects to the *powerset* of the domain, rather than to the domain itself. For instance, we could define a set-valued attribute *children* and assert $john[children \twoheadrightarrow \{mary, bob, alice\}]$. What we called "attributes" in case of the Simple O-logic will be now called *functional* (*scalar*, or *single-valued*) attributes. Details of the semantics can be found in [5, 6].

The next step is to define the class hierarchy—a common way of grouping semantically related objects together. To this end, we introduce a separate sort C of constants, called *class-objects*, and a new kind of formulas, called *IS-A atoms*. With their help we could write *john* : **student**, meaning that *john* is an instance of the class **student** $\in C$; or **student** : **person**, meaning that the class **student** is a subclass of the class **person**. Introduction of the class hierarchy and IS-A atoms enables one to write class-sensitive deductive rules, e.g.,

$$X : \textbf{basketball\_player}[salary \rightarrow high] \leftarrow X : \textbf{student}[health \rightarrow good;\ height \rightarrow H]\ \ \&\ \ H > 7ft$$

(Tall students inevitably become highly paid basketball players.)

The addition of set-valued attributes and classes gives us almost all of the functionality of O-logic as presented in [5, 6]. By removing the single-valued attributes from O-logic we obtain C-logic [1].

3 Going Higher-Order: F-logic

It was not until the higher-order syntax was introduced in [2] when a full logical formalization of object-oriented languages became possible. There are several reasons why a higher-order syntax is needed:

- In object-oriented languages, one needs to reason about classes and their instances in the same language.
- Often one has to create classes using deductive rules, just as in O-logic one does this with plain objects.
- One should be able to define the properties of classes (i.e., values of their attributes) the same way as this is done for objects.
- Higher-order syntax makes inheritance of properties from classes to subclasses easier and more elegant.
- One should be able to *browse* through the database schema in a natural way, without having to know the internal representation of the system catalogue. (This applies to relational languages as well.)

The danger in flirting with higher-order languages is that the resulting syntax may turn out to be non-computable, even for the monotonic part of the logic. To avoid this, while designing the language of F-logic we kept a constant eye on the possibility of encoding the language in first-order predicate calculus.

The higher-order syntax was achieved by breaking the distinction between classes, objects, and attributes.[4] In other words, we allow any object denotation to be construed as a reference to an object, a class, or an attribute, depending on the syntactic position of that denotation in the formula. Logical variables can now be instantiated by attribute names as well as objects. This creates the opportunity for asking queries that return sets of attributes, classes, or any other aggregation that involves these higher-order entities. For instance, to inquire about all classes to which the object *john* belongs, write:

$$?- \ john : X$$

To find all superclasses of the class student, write:

$$?- \ student : X$$

The capabilities for browsing are actually quite extensive. To get a feeling of what can be expressed, consider

$$Obj[interesting\_attrs\_of(Class) \twoheadrightarrow \{A\}] \longleftarrow Obj[A \rightarrow V] \ \& \ V : Class$$

For each class *Class* this defines an attribute *interesting_attrs_of(Class)*, whose value for any object *Obj* is the set of functional attributes (because of the single-headed arrow "$\rightarrow$") that are defined on *Obj* and return values that are instances of *Class*. We could then ask the query

$$?- \ john[interesting\_attrs\_of(person) \twoheadrightarrow \{A\}]$$

to find out which functional attributes are defined for the object *john* and return values of type *pers*

The next example shows how parametric classes can be defined in F-logic.

$$nil : list$$
$$cons(X, L) : list(T) \longleftarrow X : T \ \& \ L : list(T)$$

4 Types and Signatures

The ability to define parametric classes opens the door for parametric polymorphism, which brings in the issue of types and type-correctness. Since it is impossible to do justice here to these important issues, the motivated reader is referred to [3, 4]. Very briefly, we introduce yet another kind of logic formulas, called *signatures*, that specify the types of various attributes and methods. For instance,

$$employee[name \Rightarrow string; \ friends \Rightarrow\!\!\!> person]$$

says that every object in the class employee has a single-valued attribute *name* and a set-valued attribute *friends*. The former returns objects that belong to the class *string*, while the latter returns sets of *person*-objects. There is also a semantic type-correctness condition, which sorts ill-typed logic specifications (databases) from the well-typed ones.

[4] If necessary, this distinction can be re-established without loosing the benefits of higher-orderness. The key idea is to use a sorted language [3].

For another example, the next program defines a polymorphically typed method *append* that, being invoked on a list-object with another list-object passed as an argument, returns a concatenation of the two lists:

$$list(T)[append@list(T) \Rightarrow list(T)]$$

$$list(T)[append@nil \rightarrow L] \longleftarrow L : list(T)$$
$$cons(X, L1)[append@L2 \rightarrow cons(X, L3)] \longleftarrow L1[append@L2 \rightarrow L3] \;\&\; X : T \;\&\; L1 : list(T)$$

The first clause above is a signature that defines a parametric polymorphic type for *append*. The last two clauses actually define the *append* method. (The symbol "@" separates method names from their lists of proper arguments.) As defined, *append* is consistent with the typing constraint represented by the signature [3].

5 Conclusion

It is hoped that this short communication will motivate the reader to look into the exciting developments that have taken place in the past few years in an effort to develop a formal model for object-oriented logic languages, databases in particular. There are many issues that we did not have a chance to mention in this paper, due to the space limitation. These include structural and behavioral inheritance, the proof theory for F-logic, the programming methodology, and several others. A comprehensive report on F-logic can be obtained from the anonymous FTP account at "`cs.sunysb.edu`", file "`pub/TechReports/kifer/flogic.dvi.Z`". Other papers in this special issue advocate alternative research directions.

References

[1] W. Chen and D. Warren. C-logic for complex objects. In *Proc. of PODS*, pages 369–378, March 1989.

[2] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and schema. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 134–146, 1989.

[3] M. Kifer, G. Lausen, and J. Wu. *Logical Foundations of Object-Oriented and Frame-Based Languages.* Tech. Report 90/14, Dept. of Computer Science, SUNY at Stony Brook, July 1990.

[4] M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *Intl. Symp. on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991.

[5] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proc. of PODS*, pages 379–393, March 1989.

[6] M. Kifer and J. Wu. A logic for programming with complex objects. *JCSS*, 1991. to appear.

[7] G. Kuper and M. Vardi. A new approach to database logic. In *Proc. of PODS*, 1984.

[8] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.

[9] Z. Pawlak. Mathematical foundations of information retrieval. In *Proc. of Symp. on Mathematical Foundations of Computer Science*, pages 135–136, High Tatras, Czechoslovakia, 1973.

[10] W. Marek and Z. Pawlak. Information storage and retrieval systems: Mathematical foundations. *Theoretical Computer Science*, 1:331–354, 1976.

On Data Restructuring and Merging with Object Identity*†

Richard Hull Surjatini Widjojo
Dave Wile Masatoshi Yoshikawa

1 Introduction

Object identity, a powerful abstraction used in databases and other fields, corresponds closely to mechanisms which humans use to organize their perceptions and understanding of the physical and conceptual worlds that we live in. The emergence of semantic and object-oriented database models is now making it possible for database designers and users to utilize this abstraction in an explicit manner. The use of object identity in these models has placed a spotlight on a number of interesting, fundamental problems in data modeling, and at the same time provides the basis for resolving them. The WorldBase project [WHW89], an ongoing project at USC/Information Sciences Institute, is focused on the development of novel architectures and techniques for supporting simultaneous access to heterogeneous databases. We describe here a number of concepts and results concerning the conceptual and theoretical impact of object identity in the general contexts of data restructuring and merging, which have emerged from the WorldBase project.

In [Bee89], Beeri provides a good articulation of the distinction between *OIDs* and *values* as they arise in the database context. A value is something which has intrinsic meaning, which is universally understood (relative to the family of databases being considered); values typically include the integers, the floats, strings, booleans, and other name-based types constructed from these. In contrast, an OID has no intrinsic meaning – and derives its meaning *only* from its relationship to values and other OIDs in a given database instance. In particular, then, if an OID is considered independently from its associated database instance (e.g., in a database view or a query answer), then it conveys essentially no information other than its identity as distinct from all other OIDs. As illustrated below, this opaqueness of OIDs has impact on queries, data restructuring, and data merging (and also updates, which are not considered here).

Section 2 informally describes a formal model which encompasses OIDs and values. Section 3 presents the family of ILOG languages; these are variants of datalog which are used to specify data restructurings in the context of OIDs. Section 4 gives a more informal discussion presenting a framework for specifying data merging in the presence of OIDs. The discussion here is extremely terse; fuller exposition of many of these issues, numerous theoretical and practical results, and additional references may be found in [HY90, HY91, WHW89, WHW90, Wid90].

2 A formalism for OIDs

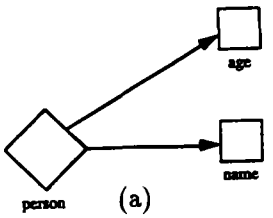
Beeri's categorization of database objects partitions the world into values and OIDs. The distinguishing feature of OIDs is that they *uniquely identify* objects from the real world — the objects being modeled. Of course, values can play this rôle in certain situations [1] – such as `Social Security number` or `part number` – but there are many situations in which it is more convenient to use pure OIDs than to artificially create printable surrogates. Examples include applications involving entity sets arising in disparate geographical, political and/or organizational contexts; objects arising in engineering design, which do not have a natural name; geographical objects; individual chips on a circuit board; and in some applications, documents, which are uniquely identified only by lengthy strings having a variety of structures.

The simple database schema of Figure 1(a) shows an *abstract class* (or entity set) `person`, which "holds"

*R. Hull is at the University of Southern California; S. Widjojo is at the Institute of Systems Science, Singapore; D. Wile is at USC/Information Sciences Institute; and M. Yoshikawa is at Kyoto Sangyo University.

†R. Hull was supported in part by NSF grant IRI-8719875; R. Hull and D. Wile in part by the Defense Advanced Research Projects Agency under DARPA grant MDA903-81-C-0335; and M. Yoshikawa in part by Science Foundation Grant # 02750298 of the Ministry of Education, Science and Culture of Japan, and a grant from the Obase Consortium.

[1] Our model formally distinguishes such values as *surrogates* for the objects, but here we only consider OIDs for unique identification.



(a)

OID	name	age
O_1	'John'	32
O_2	'Mary'	37

(b)

OID	name	age
O_5	'John'	32
O_6	'Mary'	37

(c)

Figure 1: A simple schema and two pre-instances

OIDs, with two *single-valued attributes* giving **name** and **age**, which map to character string and integer[2] types, respectively. (This, and the other schemas shown in the document are IFO$^-$ schemas as defined in [HY90]; the diagramatic conventions are essentially those of [AH87, HK87].) Parts (b) and (c) show two tabular representations of data sets that might be associated with this schema. Note that these two data sets are identical to each other, up to renaming of the OIDs. Following [AK89], we say that these two data sets are *OID-isomorphic*.

To capture the equivalence of the data sets of Figure 1(b) and (c) we call such data sets *pre-instances*, and define an *instance* to be an equivalence class of pre-instances, where two pre-instances are equivalent if there is an OID-isomorphism between them. Under these definitions, the notion of instance captures precisely the information that a database state can convey to a database user. (In contrast, a DBA may typically work directly with pre-instances.)

3 OIDs and data restructuring

We now consider the impact of OIDs on data restructuring, as it arises in database queries, schema augmentation (which arises, e.g., in derived data), and schema translation (which arises, e.g., when defining stand-alone database views). A significant impact of OIDs in these contexts stems from the axiom that OIDs independent of an associated data set have no intrinsic meaning. Thus, a query such as "List all OIDs of persons having age > 35" against the schema of Figure 1(a) yields a set of OIDs, and gives no useful information except for a cardinality. More generally, while OIDs do not carry independent meaning in query answers, they can be useful in indicating connections between different data elements (e.g., see [AK89]).

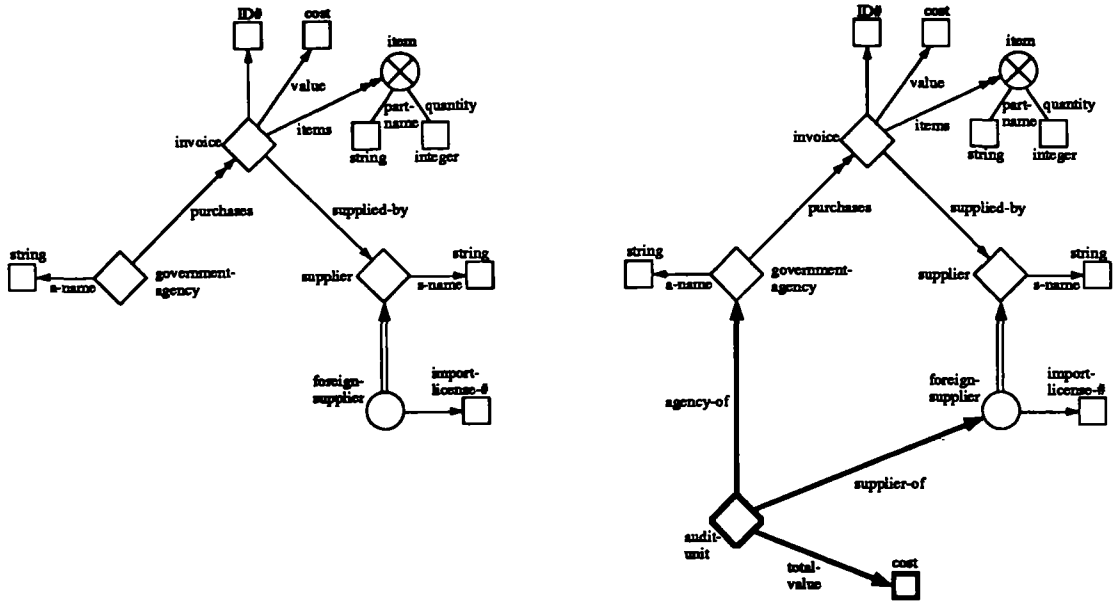
The second fundamental impact of OIDs on data restructurings is that restructurings should be capable of building, in their output, complex data sets which may involve newly "created" OIDs. Figure 2(a) shows an IFO$^-$ schema modeling hypothetical data concerning purchases made by governmental agencies. The schema includes abstract classes for `government-agency`, `invoice`, and `supplier`, and a *subclass* of `supplier` called `foreign-supplier`. The *multi-valued attribute* `purchases` maps each government agency to a set of invoices. The *aggregation* (or *tuple*) *construct* indicates that each `item` is an ordered pair, consisting of a `part-name` and a `quantity`.

Consider now the issue of augmenting this schema with the schema components highlighted in 2(b). This calls for the creation of a new entity set `audit-unit`, and two single-valued attributes. We will create an audit-unit object corresponding to each agency-supplier pair (a, s) where a has at least one invoice with foriegn supplier s. Intuitively, each audit-unit can serve as a locus for data concerning audits of such agency-supplier pairs.

ILOG is an extension of datalog, and thus uses the natural simulation of semantic models by the relational model. For an instance $\mathbf{I} = [I]$ and ILOG program P, $P(\mathbf{I})$ is defined in terms of $P(I)$, that is, in terms of pre-instances representing the instance. The following ILOG program, which has output or target relations *audit-unit*, *agency-of*, *supplier-of* and *total-of*, can be used to specify the derived data definition desired for Figure 2(b). (*int-aud-un* is used as an intermediate relation.)

$$
\begin{aligned}
\mathit{int\text{-}aud\text{-}un}(*, a, s) &\leftarrow \mathit{purchases}(a, i), \mathit{supplied\text{-}by}(i, s), \mathit{foreign\text{-}supplier}(s) \\
\mathit{audit\text{-}unit}(u) &\leftarrow \mathit{int\text{-}aud\text{-}un}(u, a, s) \\
\mathit{agency\text{-}of}(u, a) &\leftarrow \mathit{int\text{-}aud\text{-}un}(u, a, s) \\
\mathit{supplier\text{-}of}(u, s) &\leftarrow \mathit{int\text{-}aud\text{-}un}(u, a, s) \\
\mathit{total\text{-}value}(u, \mathrm{sum}\langle v \rangle) &\leftarrow \mathit{int\text{-}aud\text{-}un}(u, a, s), \mathit{purchases}(a, i), \mathit{supplied\text{-}by}(i, s), \mathit{value}(i, v)
\end{aligned}
$$

[2] Although not emphasized in this document, *types* give structural information and have immutable extents, whereas *classes* give structural information and have mutable, typically finite, extents.



(a) source schema (b) schema with augmentation

Figure 2: Schemas used to illustrate derived data and data translation

Intuitively, execution of this program on a pre-instance results in the creation of (new) OIDs for each (a, s) pair satisfying the conditions of the body of the first rule given above. The relation *int-aud-un* is used to "create" each such OID, and to "hold" its correspondence to the *witness* (i.e., tuple of values and OIDs which lead to its creation.) As with conventional datalog, variables in the rule body not occurring in the head (e.g., the variable i ranging over invoices) are viewed as existentially quantified within the body. The remaining rules are used to describe how the four components added to the schema are to be populated.

With regards to OID creation, ILOG follows the lead of [Mai86] and subsequent logics (e.g., [KW89, CW89]), by using a semantics based on Skolem functors. This permits the creation of OIDs in a systematic, set-at-a-time fashion, and forms a close tie between data restructuring and logic programming. Figure 3 illustrates the two phases of execution of the ILOG program given above. Part (a) of that figure shows a small fragment of a pre-instance for the schema of Figure 2(a). To make the presentation more intuitive, we use strings to denote OIDs representing agencies, invoices, and suppliers, and show only foriegn suppliers. Part (b) of the figure shows (part of) the output pre-instance computed by the first phase of execution. Here "OIDs" for type `audit-unit` are represented as terms constructed using the Skolem functor $f_{a\text{-}u}$; in general a distinct Skolem functor is used for each intermediate relation of a program which creates OIDs (i.e., by which a '*' is used). This output reflects the perspective on OIDs taken in [KW89, CW89], called here "exposed" semantics. (In this example, only one audit-unit OID is created for a pair (a, s), regardless of how many invoices relate a with s; this follows from the rule defining *int-aud-un*. A different ILOG program could be used to obtain other policies for OID creation.) Part (c) of the figure shows (part of) an output pre-instance of the second phase; here a "new" OID is associated with each distinct term occuring in the pre-instance computed during the first phase. This "obscured" semantics for OIDs corresponds closely to those found in most of the data restructuring languages in the literature. The instance corresponding to this pre-instance serves as the output of the program.

The full language ILOG$^\neg$ supports both recursion and stratified negation. This leads to six natural families of ILOG language, based on two dimensions: negation or no negation; and full recursion, no recursion, or "weak" recursion, which does not permit recursion through OID creation. These sublanguages correspond to various languages in the literature, for example, nonrecursive ILOG (*nrecILOG*) can be viewed as unions of conjunctive queries bundled with OID creation, and nrecILOG$^\neg$ can express the core of most schema translation languages.

There are strong similarities between ILOG and IQL [AK89] – both are relatively declarative langauges which support OID creation. We note here three fundamental differences. ILOG is essentially relational, whereas IQL supports full complex objects (in particular, sets). ILOG is essentially untyped whereas IQL is strongly typed. Finally, OID creation in ILOG is based on Skolem functors while in IQL it is based on

purchases

IRS	inv32
IRS	inv56
SS	inv11
SS	inv88

supplied-by

inv32	UK-21
inv56	PRC-5
inv11	DR-10
inv88	DR-10

(a) input

int-aud-un

$f_{a\text{-}u}$(IRS,UK-21)	IRS	UK-21
$f_{a\text{-}u}$(IRS,PRC-5)	IRS	PRC-5
$f_{a\text{-}u}$(SS,DR-10)	SS	DR-10

agency-of

$f_{a\text{-}u}$(IRS,UK-21)	IRS
$f_{a\text{-}u}$(IRS,PRC-5)	IRS
$f_{a\text{-}u}$(SS,DR-10)	SS

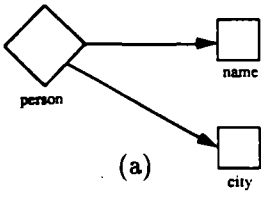
(b) output with OIDs "exposed"

agency-of

O_{37}	IRS
O_{38}	IRS
O_{39}	SS

(c) output with OIDs "obscured"

Figure 3: Partial pre-instances illustrating ILOG semantics



(a)

OID	name	city
O_5	'John'	'LA'
O_6	'Sue'	'NY'

(b)

Figure 4: The NAME-CITY schema and a pre-instance of it

something subtley different. As a result, the IQL analog of nrecILOG (i.e., positive, set-free, non-recursive IQL) is non-monotonic, while nrecILOG is monotonic.

Importantly, the Skolem functor based approach to OID creation can be used in the context of other langauge paradigms. For example, a formal semantics based on Skolem functors can easily be given for the OOAlgebra [Day89]. Thus, research concerning ILOG is "portable" to these other languages.

4 OIDs and data merging

We now turn to the impact of OIDs on database merging. This is a particularly crucial problem in the context of heterogenous databases, where users may wish to combine data from multiple autonomous databases. Under the approach of WorldBase, we assume that before merging a family of data sets, their schemas have been restructured so that they are *compatible*, in the sense that there exists a single schema which contains subschemas isomorphic to the schemas of the data sets of be merged.

The schemas of Figures 1(a) and 4(a) are compatible. Suppose now that `name` is known to be a *key* for the `person` entity class, which is universal across both databases. Then the instances represented by Figures 1(b) (or (c)) and 4(b) can be merged in an *unambiguous* fashion, to yield an instance represented by the pre-instance of Figure 5(a) (permitting `age` and `city` to be partial attributes). On the other hand, if `name` is not known to be a universal key for `person`, then the problem of merging the data of the NAME-AGE and NAME-CITY databases is ambiguous. For example, instances corresponding to either of the pre-instances of Figure 5(a) and (b) could arguably be the result of such a merge.

This example highlights two of the fundamental problems that arise in data merging: (i) determining when OIDs from two (or more) distinct databases refer to the same object "in the world", and (ii) providing a systematic formalism for specifying the "merging" of these two OIDs (or the creation of a new OID which is "linked" to these OIDs).

There is a spectrum of possibilities for determining when OIDs from different database (pre-)instances should be merged, ranging from using value-based keys, through approaches which permit OID-based keys which recursively "bottom out" with values, to approaches based on subgraph isomorphisms (e.g., identify persons from distinct databases if they both correspond to a chief executive officer of a Fortune 500 corporation whose husband serves as treasurer). A suitable formal model for specifying OID merging is yet to be developed. One possible approach is to form some kind of merger of ILOG and techniques from algebraic specifications (these come into play because of the apparent need to *equate* initially distinct OIDs.)

Another fundamental issue arises in database merging if the databases to be merged hold inconsistent

OID	name	age	city
O_{10}	'John'	32	'LA'
O_{11}	'Mary'	37	
O_{12}	'Sue'		'NY'

(a)

OID	name	age	city
O_{20}	'John'	32	
O_{21}	'John'		'LA'
O_{22}	'Mary'	37	
O_{23}	'Sue'		'NY'

(b)

Figure 5: Possible merges of data from NAME-AGE and NAME-CITY databases

information [Day83]. For example, suppose again that **name** is a universal key for the two schemas under discussion, and suppose that in one instance John is listed as having age 32, and in the other he has age 31. What value for John's age should be given in the merged database?

In WorldBase, data merging is accomplished through a two-phase process. In the first phase, the set of pairs of OIDs to be merged is determined, and in the second phase the target schema is populated using the (possibly merged) OIDs, and also all attribute and relationship data held in the source schemas. Two mechanisms are provided for dealing with inconsistent data during the second phase. The first focuses on single-valued attributes, and permits the specification of conflict resolution strategies (e.g., prefer data from one of the source databases, or compute the average value). The second mechanism provides tools to relax the integrity constraints on the target schema. In particular, a preliminary formalism is provided which permits the specification of natural combinations of the constraint sets on the source schema to serve as the constraint set on the target schema. As a simple example, the single-valued attribute **phone** in source schemas for personal and business data might be combined to form a single multi-valued attribute in the target schema, with a restriction permitting at most two phone numbers per person.

References

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

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 159–173, 1989.

[Bee89] Catriel Beeri. Formal models for object oriented databases. In *Proc. of First Intl. Conf. on Deductive and Object-Oriented Databases*, 1989.

[CW89] Weidong Chen and David S. Warren. C-Logic of complex objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 369–378, 1989.

[Day83] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 342–353, 1983.

[Day89] Umeshwar Dayal. Queries and views in an object-oriented data model. In *Proc. of Second Intl. Workshop on Database Programming Languages*, pages 80–102. Morgan Kaufmann, 1989.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers (Extended Abstract). In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1990.

[HY91] R. Hull and M. Yoshikawa. On the equivalence of database restructurings involving object identifiers. In *Proc. ACM Symp. on Principles of Database Systems*, 1991. to appear.

[KW89] M. Kifer and James Wu. A logic for object-oriented logic programming (Maier's o-logic revisited). In *Proc. ACM Symp. on Principles of Database Systems*, pages 379–393, 1989.

[Mai86] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington, D.C., August 1986.

[WHW89] Surjatini Widjojo, Richard Hull, and Dave Wile. Distributed Information Sharing using WorldBase. *IEEE Office Knowledge Engineering*, 3(2):17–26, August 1989.

[WHW90] S. Widjojo, R. Hull, and D. S. Wile. A specificational approach to merging persistent object bases. In *Implementing Persistent Object Bases*. Morgan Kaufmann, 1990.

[Wid90] Surjatini Widjojo. *Sharing Persistent Object-Bases in a Workstation Environment*. Ph.D. Thesis, Computer Science Department, University of Southern California, August, 1990.

Data Structures and Data Types for Object-Oriented Databases

Val Breazu-Tannen, Peter Buneman and Atsushi Ohori*

1 Introduction

The possibility of finding a static type system for object-oriented programming languages was initiated by Cardelli [Car88,CW85] who showed that it is possible to express the polymorphic nature of functions such as

```
fun age(x) = thisyear - x.year_of_birth
```

which may be regarded as a method of the "class" of record values that contain a numeric age field. It is possible both to integrate this form of record polymorphism with the parametric (universal) polymorphism and also to express a number of object-oriented programming paradigms by combining higher-order functions with field selection. Since then a number of alternative schemes [Wan87,Sta88,OB88,JM88,Rem89,HP91] have been developed that include the possibility of type inference and the use of abstract types [OB89]. The extent to which these typing schemes give a satisfactory account of all aspects of object-oriented languages remains an open question, and it may therefore be premature to complicate the picture by introducing database concepts. Nevertheless, if we are to treat databases of any kind (object- oriented or otherwise) properly in typed programming languages there are certain issues that must be resolved, and it is these that we shall briefly investigate in this paper.

Unlike the languages associated with the relational data model which have simple and more or less coincident operational and denotational semantics, the authors know of no equivalent formulation of an object-oriented data model. While several papers, e.g. [ABD+89] describe certain *desiderata* of object-oriented databases, a consensus has yet to emerge on a formalism for an object-oriented data model, nor is there yet an adequate explanation of what is fundamentally new about such a model. The issues we discuss in this paper are relevant to object-oriented databases because they are of general concern in languages [Sch77,ABC+83,ACO85] that integrate database structures with their type systems, and object-oriented databases surely fall into this category.

We shall be mainly concerned with operations on records and some "bulk" data type such as sets. Any database programming language [AB87] must surely be capable of expressing a function such as

```
fun wealthy(S) = select x.Name
                 from x <- S
                 where x.Sal >= 100,000
```

The syntax here is taken from Machiavelli [OBBT89], but very similar definitions are to be found in object-oriented languages such as O_2 [LRV88]. We would like a type system to express exactly what is required of the argument S in order for the function wealthy to be well defined. S contains *records* (perhaps objects) with appropriate properties. S must be a set (or some other bulk type such as a bag or list.) Finally we must allow that in some databases S may be *heterogeneous*: the individual records may not all have the same structure. These three demands on a programming language are the issues we discuss in this paper.

*Authors addresses: Buneman and Breazu-Tannen, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-6389, USA; Ohori, Kansai Laboratory, OKI Electric Industry, Crystal Tower, 1-2-27 Chuo-ku, Osaka 540, Japan. Breazu-Tannen was partially supported by grants ONR NOOO-14-88-K-0634 and NSF CCR-90-57570. Buneman was partially supported by grants ONR NOOO-14-88-K-0634, NSF IRI-86-10617 and by a UK SERC visiting fellowship at Imperial College, London

2 Operations on records

In [OBBT89] the function `wealthy` is given the type

```
{[('a) Name:'b, Sal:num]} -> {'b}
```

The `'a` and `'b` are *type variables* and may, subject to restrictions discussed below, be instantiated by any type. The notation `{'b}` describes the type of sets of values of type `'b`, and the notation `[('a) Name:'b, Sal:num]` describes the restriction that an instance of `'a` must be a record type that contains the fields `Name` : σ and `Sal:num` where σ is any instance of `'b`. For example

```
{[Name:string, Sal:num, Weight:num]} -> {string}
{[Name:[First:string, Last:string], Sal:num]} -> {[First:string, Last:string]}
```

are legal instantiations of the type of `wealthy`.

Using such a syntax it is not only possible to express the exact polymorphic type of a function like `wealthy`; it is possible to *infer* a type by means of an extension of ML's type inference system. If the only operations on records were record formation and field selection, the necessary techniques are now well established (in fact the approaches given in [Wan87,Sta88,OB88,JM88,Rem89] would agree.) The differences arise when we add operations that extend or combine records, and this is where databases place an unusual demand on the type system. An operation common in databases is to *join* two records on consistent information. For example `[Name='Joe', Age=21]` and `[Name='Joe', Sal=30,000]` join to form `[Name='Joe', Age=21, Sal=30,000]`. On the other hand there is no type that can be given to the join of `[Id=1234]` and `[Id='A123']`. This join can be extended to sets of records i.e. relations, and in fact to arbitrary structures on which equality is provided, to define the *natural join* of complex objects [Oho90b]. It is arguable that natural join is needed in a database programming language, but even if it is not, a very similar typing rule is needed for the intersection of heterogeneous sets (see below).

There is a well-known result [Mil78] that underlies the polymorphic type system of ML that every expression has a *principal* type scheme, i.e. every possible ground type for an expression can be obtained by instantiating the type variables of its principal type. If we add the typing rules for record formation and field selection:

$$\text{(RECORD)}\quad \frac{\mathcal{A}\rhd e_1 : \tau_1,\ \ldots,\ \mathcal{A}\rhd e_n : \tau_n}{\mathcal{A}\rhd [l_1{=}e_1,\ldots,l_n{=}e_n] : [l_1:\tau_1,\ldots,l_n:\tau_n]} \qquad \text{(DOT)}\quad \frac{\mathcal{A}\rhd e : [\ldots, l:\tau, \ldots]}{\mathcal{A}\rhd e.l : \tau}$$

we retain the principal typing property [Oho90a]. However, the rule for join is unusual in that it can only be used provided a "side condition" is satisfied:

$$\text{(JOIN)}\quad \frac{\mathcal{A}\rhd e_1 : \delta_1 \qquad \mathcal{A}\rhd e_2 : \delta_2}{\mathcal{A}\rhd \mathbf{join}(e_1,e_2) : \delta} \qquad \text{if } \delta = \delta_1 \sqcup \delta_2$$

The requirement that any ground type also satisfy the side conditions means that we need to relax the notion of the principal typing property to include these conditions; nevertheless it is still decidable whether a given expression has a type, and by suitably delaying the checks for satisfaction, the process of type inference can be made efficient and to operate interactively.

3 Operations on sets

The operations of the relational algebra suggest one way in which operations on sets may be added to a programming language. While these are adequate for a large number of database applications, there are a number of useful operations, such as transitive closure of a binary relation, that cannot be expressed with the relational algebra alone. Moreover, there is no way of expressing the cardinality, sum, or other aggregate operations on a set. Relational query languages provide these as special operations, but there is no general way to construct new aggregate operations.

The problem is this: the relational algebra provides us with an adequate set of operations for mapping sets of tuples (records) into sets of tuples, but provides us with no way of moving outside this domain; we cannot expect the relational algebra to produce a set of sets or an integer. One could get rounds this by adding a *choose* operator, which picks arbitrarily an element of a set, and using general recursion to

program functions mapping sets to other types. However, *choose* introduces a nondeterministic semantics, and makes it difficult to ensure that our programs are well-behaved. A better approach, we claim, is to use *structural recursion* as the general technique for carrying sets into other structures. As opposed to general recursion, which most of the times requires destructors like *choose*, this form of programming works by matching arguments against data type constructors. One form of structural recursion on sets is given by the combinator Φ which takes $E : \beta$, $F : \alpha \to \beta$ and $U : \beta \times \beta \to \beta$ to the unique $\Phi(E, F, U) : \{\alpha\} \to \beta$ satisfying

$$\Phi(E, F, U)(\emptyset) = E$$
$$\Phi(E, F, U(\{x\}) = F(x)$$
$$\Phi(E, F, U)(s_1 \cup s_2) = U(\Phi(E, F, U)(s_1), \Phi(E, F, U)(s_2))$$

provided that on the range of $\Phi(E, F, U)$, U is associative and E is an identity for U (a monoid structure), and moreover that U is commutative and idempotent. This is similar to the *pump* operator of FAD [BBKV88] and the *hom* operator of Machiavelli [OBBT89], except that in those languages the requirement of idempotence is dropped and the requirement that the sets s_1, s_2 be disjoint in the third clause is added. *Pump* and *hom* have a natural denotational semantics, but their operational semantics is contrived. The evaluator must evaluate sets eagerly and then do time consuming dynamic tests for equality of values. Of course, this rules out working with sets of functions for example. Even for sets of, say, integers, mapping a function over a disjoint union may yield a non-disjoint one, which fed into *hom* would yield a run-time error. One would like to obtain statically an assurance that the program goes through, but it seems that only a few very simple programs can be shown correct in this sense. On the other hand *pump* and *hom* can be implemented in Φ style, by converting sets to *bags* and then doing structural recursion on those.

Appropriate uses of Φ are, for example, $\Phi_\cup(F) = \Phi(\emptyset, F, \cup)$ and $\Phi_\wedge(P) \doteq \Phi(\text{true}, P, \wedge)$ where $F : \alpha \to \{\gamma\}$ and $P : \alpha \to \text{bool}$. Using these, we can construct the following functions on sets:

$$\text{map } f = \Phi_\cup(\lambda x.\{fx\})$$
$$\text{pairwith } s\ x = \text{map } (\lambda y.(x, y))\ s$$
$$\text{cartprod}(s_1, s_2) = \Phi_\cup(\text{pairwith } s_2)(s_1)$$
$$\text{powerset } = \Phi(\{\emptyset\}\ ,\ \lambda x.\{\emptyset\} \cup \{\{x\}\}\ ,\ \lambda(s_1, s_2).\ \text{map } \cup\ \text{cartprod}(s_1, s_2))$$

(checking the commutative-idempotent monoid requirement for the use of Φ in the last definition is quite interesting).

The denotational and operational semantics as well as an appropriate logic for reasoning about programs that compute with structural recursion over bulk data types such as lists, bags and sets is studied in [BS91], where transformations to other presentations of these datatypes are also given. In [BBN91], it is shown how to compute transitive closure efficiently with structural recursion, and it is noted that relational algebra can be characterized using restricted forms of structural recursion: the expressions of relational algebra are semantically equivalent to precisely those expressions that can be constructed using the structural recursors $\Phi_\cup$ and $\Phi_\wedge$ together with elementary operations (concatenation, projection and conditionals) on tuples.

4 Heterogeneous collections

The ability to deal with heterogeneous collections is claimed [Str87] as an important feature of object-oriented programming, and we believe it is of special importance in object-oriented databases, where it appears to be the only way to reconcile two natural views of inheritance [BO90]. Before looking at this issue we should remark that we have so far been working in a framework of *typed* languages. These are languages in which the only meaningful expressions are those that have a (declared or inferred) type. In such a language `3 + "cat"` is not a program because it has no type. Compare this with the situation in "dynamically typed" languages in which such expressions can be evaluated, but may yield run-time type errors. In any persistent programming language [AB87] it is desirable, for safety, to maintain a structure that describes the type of a database along with the database. However, in order to reason about these external types in a typed language requires some extra apparatus.

The need for this is seen in any language that has some form of subtype rule in conjunction with a bulk data type such as lists. If l is an expression of type *list*(*Person*) and e is an expression of type *Employee*, the expression cons(e, l) that "inserts" e into l also has, because of the subtype rule, *list*(*Person*). The expression

head(cons(e, l)) now has type *Person*, and we have "lost" some of the structure of e; more generally we can no longer use the equation e = head(cons(e, l)) to reason about our programs. In [BO90] we have proposed an extension of the *dynamic* types [ACPP89] in which values in a programming language are "views" that express the partial type of some completely typed object.

The way we achieve this is to incorporate into our type system a distinction between the *type* of an object and its *kind*. In object-oriented terminology the former specifies the class of an object and hence its exact structure, while the latter specifies that certain methods are available. In order to incorporate assertions on kinds of objects in the type system, we introduce a new form of assertion $e : \mathcal{P}(\kappa)$ denoting that e has the kind κ. For example, e:$\mathcal{P}$(<Name:string, Age:num>) means that at least Name and Age fields are available on e. <Name:string, Age:num> describes a *kind*, which we can think of as a set of types – the set of record types that contain Name:string and Age:num components.

Kinds are most useful in conjunction with heterogeneous collections, which may not have a uniform type, but may have a useful kind. For example, e:{$\mathcal{P}$(<Name:string, Age:num>)} means that e is a set of records, each of which has at least a Name and Age field, and therefore queries involving only selection of these fields are legitimate. To construct such a heterogeneous collections of uniform kind, an operation **filter** κ (S) is defined which selects all the elements of S which have the fields specified by κ and makes those fields available, i.e. **filter** κ (S):{$\mathcal{P}$(k)}.

An advantage of this approach is that it reconciles the database "isa" hierarchies (with extent inclusion) with the hierarchies of object-oriented languages (with method sharing.) To show this, let us assume that the following names have been given for kinds:

```
PersKind   for  <Name:string, Address:string>
EmpKind    for  <Name:string, Address:string, Sal:num>
```

Also suppose that DB is a set of type {$\mathcal{P}$(any)}. The meaning of any is the set of all possible types, so that we initially have no information about the structure of members of this set.

Since kinds denote sets of types, they can be ordered by set inclusion. In particular, Empkind is a "sub-kind" of PersKind. From this, the inclusion **filter** EmpKind (S) $\subseteq$ **filter** PersKind (S) will always hold for any heterogeneous set S. This means that the "data model" (inclusion) inheritance is *derived* as a static property from an ordering on kinds rather than being something that must be achieved by the explicit association of extents with classes with dynamic maintenance of extents. Moreover, object-oriented (method sharing) inheritance is also derived from a polymorphic type of a method. For example, the type inference method we have described in section 2 guarantees that any polymorphic function applicable to $\mathcal{P}$(PersKind) is also applicable to $\mathcal{P}$(EmpKind). Thus, we achieve the desired coupling of the two forms of *is-a* in a static type system.

5 Conclusions

We have attempted to show that typed languages are a natural medium for many aspects of database programming languages. There are certain topics such as object identity, abstract types, views (and the interaction between these) that require further investigation. However we are confident that these can be resolved and that object-oriented databases will be best understood in the same framework of typed languages.

References

[AB87] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, June 1987.

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), 1983.

[ABD+89] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrick, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. First Deductive and Object-Oriented Database Conference*, 1989.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *Transaction on Database Systems*, 10:230–260, 1985.

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. ACM Symp. on Principles of Programming Languages*, 1989.

[BBKV88] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 97–105, 1988.

[BBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. Unpublished Manuscript, University of Pennsylvania, 1991.

[BO90] P. Buneman and A. Ohori. A type system that reconcile classes and extents. Technical report, University of Pennsylvania, 1990.

[BS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. To appear in *Proc. International Conference on Automata, Languages and Programming (ICALP)*, 1991.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proc. ACM Symp. on Principles of Programming Languages*, 1991.

[JM88] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, 1988.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. ACM SIGMOD Conference*, pages 424–434, 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[OB89] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proc. ACM OOPSLA Conference*, pages 445–456, 1989.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. ACM SIGMOD conference*, pages 46–57, 1989.

[Oho90a] A. Ohori. Extending polymorphism to records and variants. Technical Report, University of Glasgow, 1990.

[Oho90b] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, 76:53–91, 1990.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symp. on Principles of Programming Languages*, 1989.

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. *Transactions on Database Systems*, 5(2), 1977.

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[Str87] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1987.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. Symposium on Logic in Computer Science*, pages 37–44, 1987.

Foundations of the O_2 Database System

C. Lécluse and P. Richard

Altaïr, BP 105
78153 Le Chesnay Cedex, France

1 Introduction

In this paper, we describe the data model of the O_2 system as it is implemented. We first implemented, in December 87, a throw away prototype [BBB*88] in order to test and show the functionalities of the system. This gave us a lot of feed back and we completely redesigned the system, its data model [LR89], its language and its architecture. The first prototype convinced us that the object-oriented approach was a good choice for databases. However, we also deduced from this experience that a lot of points should be carefully designed otherwise object-oriented databases might not reach their goals and might even be a step backward. Among the new features which we think are necessary to implement, let us quote: (i) complex values together with objects, (ii) names for objects and values with an automatic persistence mechanism attached to names, (iii) the list type constructor, (iv) the possibility of separating classes and method definitions from the implementation, (v) the need for a design mode where emphasis is made on dynamicity and evolution and an execution mode where performance are crucial. The O_2 data model relies on two kinds of concepts: complex values, on which we can perform a predefined set of primitives, and objects which have an identity and encapsulate values and user defined methods. Values have types which specify their structure and objects belong to classes.

2 Values and objects

The presentation of this section has been largely influenced by the works of [LR89] and also [AK89]. We suppose given: (i) A set of atomic types names {integer, string, float, boolean} and their corresponding domains, $D_{integer}$, D_{string}, D_{float}, $D_{boolean}$, which are pairwise disjoint. The set **D** of basic values is the union of these basic types domains. (ii) A set **A** of symbols called *attributes*: **age**, **name**, ... (iii) A set **I** of *object identifiers*: #32, #765, ... (iv) A set of *class names* **C**.

In the following, we shall use capitals for class names, typewriter for attribute names and # followed by numerals for object identifiers.

Definition 1 Let I be a subset of **I**. A *value over* I (or just value if I is understood) is recursively defined as follows:

- The symbol *nil* is a value.
- Every element of **D** or I is a value.
- If $v_1, .., v_n$ are values then $[a_1 : v_1, .., a_n : v_n]$ is a (tuple) value. [ ] is the empty tuple value.
- If $v_1, .., v_n$ are distinct values then $\{v_1, .., v_n\}$ is a (set) value. {} is the empty set value.
- If $v_1, .., v_n$ are values then $< v_1, .., v_n >$ is a (list) value. $< >$ is the empty list value.

We denote $\mathbf{V}(I)$ the set of all values over I and $\mathbf{V} = \mathbf{V(I)}$. An *object* is the association of an object identifier of **I** with a value of **VI**. We note **O** the set of all objects. Classically, in object-oriented data models, every piece of information is an object. In the O_2 data model, we allow both the concept of object and value. This means that, in the definition of an object, the component values of this object do not necessarily contain objects, but also other values. Here are some examples of values and objects:

[name: "Eiffel_tower",
address: [city: #432,
street: "Champ de mars"],
description: "Paris Monument",
admission_fee: 25]

{#23, #54}

#23 → [name: "Eiffel_tower",
address: [city: #432,
street: "Champ de mars"],
description: "Paris Monument",
admission_fee: 25]

#432 → [name: "Paris",
country: "France",
population: 2.6,
monuments: {#23, #54}]

In object-oriented database systems, this value is classically a tuple or a set of objects since databases must provide flexible management of large sets of data. However, this value is always a flat value, as it can only contain identifiers of other objects, and not directly other complex values. This limitation is exactly like the limitation of relational systems which has motivated the introduction of nested relations and complex objects. In O_2, we provide the user with the possibility of manipulating, not only objects, but also *values* as in standard programming languages or in the so-called complex objects[1] languages. Of course, complex (nested) structures can always be modeled through the use of identifiers but we think that this solution is awkward, like the modeling of nested relations with surrogates in relational systems. The address of the object #23 above could have been written using an intermediary objet. However, this address is conceptually nothing else than a pair of strings and is totally local to the object #23.

3 Types and classes

Definition 2 Let C be a subset of **C**. We call *types over* C (or just types, if C is understood), the expressions constructed as follows:

- The symbol **any** is a type.
- The atomic types **integer**, **float**, **boolean**, and **string** are types.
- The class names of C are types.
- If $t_1, ..., t_n$ are types, then $[a_1 : t_1, .., a_n : t_n]$ is a type.
- If t is a type then {t} is a type.
- If t is a type then <t> is a type.

We shall note $\mathbf{T}(C)$ the set of all types and $\mathbf{T} = \mathbf{T(C)}$.

The following are examples of types. Notice that these two types are referencing classes Monument and City.

[name: string,
address: [city: City,
street: string],
description: string,
admission_fee: integer]

[name: string,
country: string,
population: float,
monuments: {Monument}]

[1] which are not objects in the object-oriented terminology but rather complex values.

4 Class hierarchy and subtyping

Inheritance is a central concept in object-oriented (database) systems. It allows the user to derive new classes from existing classes by refining their properties. For example, the Monument class can be specialized into a Historical_monument class. We say that the Historical_monument class inherits from the Monument class. Following the abstract data type theory, the concept of subtype is based on the idea that a type is a form of behavior and a subtype is a compatible specialization of the behavior. A class hierarchy is made of two components: a set of class names with types associated to them, and a subclass relationship. The type associated to a class describes the structure of the objects which are instances of the class. The subclass relationship describes the (user-defined) inheritance properties between classes.

Definition 3 A *class hierarchy* is a triple $(C, \sigma, \prec)$ where C is a *finite* set of class names, σ is a mapping from C to $\mathbf{T}(C)$, and $\prec$ is a strict partial ordering among C.

The type σ(c) is the structure of the class of name c. We derive a *subtyping* relationship $\leq$ from the subclass relationship as follows:

Definition 4 Let $(C, \sigma, \prec)$ be a a class hierarchy, the subtyping relationship $\leq$ on T is the smallest partial ordering which satisfies the following axioms:

- $\vdash$ c $\leq$ c', for all c, c' in C such that c $\prec$ c'.
- $\vdash [a_1 : t_1, .., a_n : t_n, .., a_{n+p} : t_{n+p}] \leq [a_1 : s_1, .., a_n : s_n]$, for all types t_i and s_i, i=1, ..., n such that $t_i \leq s_i$.
- $\vdash$ {t} $\leq$ {s}, for all types s and t such that s $\leq$ t.
- $\vdash <$ t $> \leq <$ s $>$, for all types s and t such that s $\leq$ t.
- t $\leq$ **any**, for all types t.

The first rule just expresses that subclasses are subtypes. We then have one rule per type structure. Notice that we can refine tuples by refining some fields or by adding new ones. We do not allow two related classes (c $\prec$ c') to have arbitrary associated types. The type associated to a class describes the internal structure of the instances of the class. An instance of a class (say Employee) being also an instance of its superclass Person, we want the instances to share common structures. More precisely, if c and c' are two related classes (c $\prec$ c'), then we want to ensure that σ(c) $\leq \sigma$(c'). The following definition characterizes consistent class hierarchies.

Definition 5 We say that the class hierarchy $(C, \sigma, \prec)$ is consistent iff for all classes c and c', if $c \prec c'$ then $\sigma(c) \leq \sigma(c')$.

Methods are associated to classes and define the behavior of objects of the corresponding class. One important feature of the object-oriented paradigm is the *encapsulation*. The objects of a class can only be manipulated using the methods associated to this class. A method can be seen as a function from a source domain to a range domain. For example, the method get_name: Monument $\rightarrow$ string can be applied to objects belonging to class Monument and the result will be a value of type string. A method is represented in the O_2 model by a signature. A method *signature* on C is an expression $m : c \times t_1 \times \ldots \times t_n \rightarrow t$, where m is the name of the method, and c, $t_1 \ldots t_n$ are types over C. We also impose that the first type c is a class name. The first type of a method signature is the class to which the method is attached and it is called the *receiver class* of the method. An important feature of object-oriented systems is the notion of *overloading* and *late binding*. A method m can be defined with *the same name* in several classes. Given an object and a method name, the code to be executed is determined at run-time by searching for a method of that name in the class hierarchy. Of course, the redefinitions of a method in subclasses of a class must follow some typing rules in order to avoid typing inconsistencies (see Definition 8).

5 Database schema

Definition 6 A *database schema* is a 5-tuple $S = (C, \sigma, \prec, M, G)$ such that: (i) $(C, \sigma, \prec)$ is a consistent class hierarchy, (ii) M is a set of method signatures on C, (iii) G is a set of names with a type associated to each name.

A database schema models both the structural and the behavioral parts of the database. Its main component is the class hierarchy and the method signatures which describe the programming interface of the instances of the classes. The names of G are the entry points in the database. They serve as handles for some objects (or values) which are of a particular importance. In many object-oriented systems, the database entry points are the extensions of the classes. We want to be more general and define entry points which are arbitrary values or objects. Moreover, as database entry points, the names of G (global names) are also used to define the persistence semantics with the following rules (i) every object or value attached to a global name is persistent, (ii) every object or value attached to a persistent object or value is also persistent. The global names are the roots of the persistence mechanism. In order to define a notion of schema consistency, we first state the following working definition:

Definition 7 Let S be a database schema. If m is a method name and c a class of C, we say that *m is defined in c* if there is a signature m : c $\times$ $t_1 \ldots t_n$ $\rightarrow$t in M. We say that *m is reachable from c* if there is (at least) one superclass of c in which m is defined.

The notion of reachability above defines method inheritance. Indeed, if we view a method as a function, this function is defined in a class c but is reachable from (can be applied to) any subclasses of c. Methods inheritance must also follow some typing rules in order to avoid inheritance conflicts. If two non comparable classes defining a method m have a common subclass (say c"), then the method m can be inherited from any of these superclasses. The semantics of method inheritance must ensure that the application of a method to an object is uniquely defined. We now define the notion of a consistent database schema.

Definition 8 A database schema $S = (C, \sigma, \prec, M, G)$ is *consistent* if and only if it satisfies the following properties: (i) if c $\prec$ c' and m: c $\times$ $t_2 \ldots t_n$ $\rightarrow$t and m: $c' \times t'_2 \ldots t'_n$ $\rightarrow$t' are in M, then $t_i \leq t'_i$ and t $\leq$ t' (covariant condition), (ii) if there are classes c and c' having a common subclass c", with a method of name m defined on both c and c', then there is another subclass c"' of c and c' *of which c" is a subclass* in which m is also defined.

The first property ensures that the method overloading is done with compatible signatures, and the last one eliminates the multiple inheritance conflicts. In the following, we shall always consider consistent database schemas.

6 Instances of a database schema

In order to define the instances of a database schema, we have to define the interpretation of a type. This interpretation is defined, given an *oid assignment* [AK89] which describes the instances of the classes in the hierarchy.

Definition 9 Let $(C, \sigma, \prec)$ be a subclass hierarchy. An *oid assignment* is a function π mapping each class name on a set of object identifiers and such that for pairs of classes (c, c') in C verifying c $\prec$ c', we have π(c) $\subseteq$ π(c').

The constraint satisfied by an oid assignment maps the inheritance links from the classes to the instances. It means that an employee is a person and that a hotel is a monument.

Definition 10 Let S be a (consistent) database schema. Given an oid assignment, the *interpretation* dom(t) of a type t in $\mathbf{T}(C)$ is defined as follows.

- If I = $\cup\{\pi(c) \mid c \in C\}$ then dom(any) = V(I).

- For every atomic type d, dom(d) is the domain associated in Subsection 2.
- dom(c) = {nil} $\cup$ π(c) for all c $\in$ C.
- dom([$a_1 : t_1, \ldots, a_n : t_n$]) = {[$a_1 : v_1, \ldots, a_n : v_n, a_{n+1} : v_{n+1}, \ldots, a_{n+m} : v_{n+m}$] | $v_i \in$ dom(t_i), i=1, ..., n}.
- dom({t}) = {{$v_1, \ldots, v_n$} | $v_i \in$ dom(t), i=1, ..., n}.
- dom (<t>) = {< $v_1, \ldots, v_n$ >| $v_i \in$ dom(t), i=1, ..., n}.

The domain function is built in the usual way. Condition 1 expresses that the only valid object identifiers are those of the existing instances. We follow the Cardelli approach [C84] for the domain of tuple types and allow tuple values with extra attributes. This definition leads to a domain inclusion semantics for subtyping, as shown in the following lemmas which generalize the inclusion of domains from classes to arbitrary types.

Lemma 1 Let $S = (C, \sigma, \prec, M, G)$ be a consistent database schema. For all types t and t' in T(C) such that t $\leq$ t' we have dom(t) $\subseteq$ dom(t').

Lemma 2 Let $S = (C, \sigma, \prec, M, G)$ be a consistent database schema. For all types t and t' in T(C), t $\leq$ t' if and only if dom(t) $\subseteq$ dom(t') *for all* oid assignment π.

We now define the instance of a database schema.

Definition 11 An *instance* of a database schema S consists of a 4-tuple $(\pi, \nu, \delta, \alpha)$, where:

- π is an oid assignment for the schema and I = $\cup\{\pi(c) \mid c \in C\}$.
- ν is the mapping from object identifiers to the associated values, that is ν is a function from I to **V**(I). This function defines the value associated to all the database object identifiers. Of course, these values must be consistent with the type of the corresponding classes, and we impose that: $\forall j \in \pi(c)$, $\nu(j) \in dom(\sigma(c))$.
- δ is an assignment for each method signature in M, such that δ (m: w $\rightarrow$ t) $\in dom(w)^{dom(t)}$. If the method m is overloaded with two different signatures w $\rightarrow$ t and w' $\rightarrow$ t', such that w$\leq$w' and t$\leq$t', then we impose that the functions δ (m: w $\rightarrow$ t) and δ (m: w' $\rightarrow$ t') agree on dom(w).
- α is a function associating each name of G, of type t, on a value of dom(t). This value is the value currently assigned to the global name.

Acknowledgements

The work on the database programming language and the data model benefited from help and feed back from many of the Altaïr members. We are particularly indebted to our colleagues in the language team and the system team. We also especially acknowledge F. Velez who participated in the design of a first version of the model and S. Abiteboul and P. Kanellakis who defined an object-oriented data model [AK89] from which we took inspiration.

References

[AK89] S. Abiteboul and P. Kanellakis, " Object Identity As a Query Language Primitive", *in proc. of ACM-Sigmod, 1989.*

[BBB*88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard and F. Velez, "The design and Implementation of O_2, an Object-Oriented Database System", *in Advances in Object-Oriented Database Systems Springer-Verlag, September 1988.*

[C84] L. Cardelli, "A Semantics of Multiple Inheritance", Semantics of Data Types, *Lecture Notes in Computer Science, 1984.*

[LR89] C. Lécluse and P. Richard. "Modeling Complex Structures in Object-Oriented Databases", *in proc of the PODS 89 Conference, Philadelphia, March 29-31, 1989.*

UPDATING THE SCHEMA OF AN OBJECT-ORIENTED DATABASE

(Extended abstract)

Alberto Coen-Porisini (*)
Luigi Lavazza (**)
Roberto Zicari (*)

(*) Politecnico di Milano, Dipartimento di Elettronica, Piazza Leonardo da Vinci, 32 - 20133 Milano, Italy
e-mail: relett15@imipoli.bitnet

(**) Cefriel, via Emanueli 15, 20100 Milano, Italy

1. INTRODUCTION

Schema evolution is a concern in object-oriented systems because the dynamic nature of typical object-oriented database applications calls for frequent changes in the schema [Pan88].

Therefore, updating the schema is an important facility for object-oriented databases. However, updates should not result in inconsistencies either in the schema or in the database. Ensuring the consistency for an object-oriented database is a difficult task. This difficulty is due to the richness of the object-oriented data model and to the use, in most systems, of an imperative language to implement methods.

One of the authors of this paper began addressing the problem in 1989 for a specific object-oriented database system: O_2 developed by GIP Altaïr.

We assume in the rest an object-oriented database system with the characteristics illustrated in the next subsection, but we do not refer to a specific oodbms.

1.1 Basic Assumptions

We consider a database system with both the notion of types and classes. Instances of a class are objects which encapsulate data and behavior. Instances of a type are values. To every class is associated a type describing the structure of the class instances. Types can be complex; they are created recursively using atomic types (integer, boolean, etc...), class names, and the set, list, and tuple constructors. Objects have a unique internal identifier and are encapsulated, their values are not directly accessible and they are manipulated by methods. Method definition is composed of two parts: a signature, that is the type of the arguments (if any) and the type of the result (if any), and a body which contains the code of the method. We assume an imperative language for specifying the body of methods. (This contrasts with other object-oriented database systems based on "pure" object-oriented paradigms such as SmallTalk). Methods are attached to classes and therefore they are part of the schema. We consider multiple inheritance and assume a semantics for inheritance based on the notion of subtyping. Inheritance between classes defines a class hierarchy.

A schema is composed of a set of classes related by inheritance which follows the type compatibility rules of subtyping, and/or by composition links, plus a set of methods. Class attributes and methods are identified by name. The system uses late binding and allows polymorphism. We also assume the system offers a compile-type checker to statically detect as many illegal manipulations as possible of objects and values. Examples of such OOL languages are the Eiffel programming language [Mey88], and the schema definition language of the O_2 object-oriented database system [LR89].

1.2 Schema updates: What is the problem?

Informally, the problem with schema updates can be stated as follows: We want to change the structure and behavioral part of a set of classes without resulting in run-time errors, "anomalous" behavior and any other kinds of uncontrollable situations. In particular, we want to assure that the semantics of updates are such that when a schema is modified, it is still a consistent schema.

Consistency can be classified as follows:

1. *Structural consistency*. This refers to the static part of the database. An object-oriented database is structurally consistent if:

i) its class structure is a direct acyclic graph (DAG);

ii) its attribute and method name definitions, attribute and method scope rules, attribute types and method signatures are all compatible. In particular we assume a *covariance condition* to ensure that method overloading is done with compatible signatures;

iii) no multiple inheritance conflicts (also denoted as name conflicts) must occur.

This work has been partially supported by CNR - Progetto Finalizzato Sistemi Informativi e Calcolo Parallelo.

2. *Behavioral consistency*. This refers to the dynamic part of the database. An object-oriented database is behaviorally consistent if execution of methods does not result in run-time errors or unexpected results. Checking the signature of a method is not sufficient to ensure behavioral compatibility of the method. There are two distinct notions of behavioral inconsistencies: *method's failure*, i.e. run-time errors, and *method's change of behavior*, i.e. no run-time errors occur, but the result of the method is different than the one expected. In the rest of this paper we will only consider the first case.

2. SCHEMA UPDATES PRIMITIVES

For reason of space we do not report the syntax and semantics of the full list of schema update primitives we have defined for changing the schema. A minimal set of schema updates is the following:

1. *Add an attribute in a class, Delete an attribute in a class,*

2. *Add a method in a class, Delete a method in a class,*

3. *Add a class, Delete a class, Make a class a superclass (subclass) of another class, Remove a class from the superclass (subclass) list of a class, Change the name of a class.*

Updates can be further classified as type-preserving and non-type preserving.

It is also worth noticing how the notion of *completeness* of a set of basic updates at schema level, that is whether the set of basic updates subsumes every possible type of schema change, is not necessarily the same one at data base instance level. For example, consider the update: *change the name of a class*, at schema level this is equivalent to deleting the class and then adding a new class. However this obviously does not hold at (object) instance level.

There are different ways to define the semantics of updates, especially of the ones listed at point 3. One way to go is to define very basic updates with a default semantics, the other is to define higher-level *parametrized* primitives allowing the designer to define his/her own semantics. The following simple example shows this point. Consider the schema of Figure 1 composed of three classes: Person, PhD, and Employee (with associated types Ta, Tc, Tb).

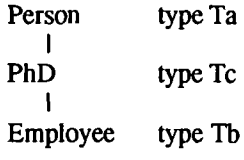


Figure 1

Suppose we perform the update: *remove class PhD from the superclass list of Employee*. The effect of this update is in disconnecting the class Employee from the DAG (PhD is the only superclass of Employee). To preserve schema consistency, the class Employee has to be connected to some other class(es) in the DAG. There are two possibilities:

- Class Employee is made a direct subclass of direct superclasses of PhD (class Person in the example);

- Class Employee is made a direct subclass of the root class of the DAG: OBJECT.

Moreover, we need to define what happens to the type of Employee Tb. There are again two possibilities:

- class Employee loses all attributes inherited from PhD;

- class Employee does not lose attributes inherited from PhD; attributes which were inherited become locally defined in Employee.

Same considerations hold for methods as well.

The use of parametrized update operators allows the definition of all these different update semantics. A more precise definition of the set of basic schema updates primitives and of the parametrized ones is reported in [Zic90], [Zic91a], [Zic91b].

3. ENSURING STRUCTURAL CONSISTENCY

An update to a schema is a mapping which transforms a schema S into a (possibly) different schema S'. Schemas S and S' have to be structurally consistent. The semantics of the schema update primitives will have to ensure *at least* that structurally consistent schemas are produced as a result of an update.

The approach is to use a graph-theoretic tool set to enforce necessary conditions for the structural consistency of a schema: name conflicts, type conflicts, requirements for signatures, and cycle conflicts. This is obtained by mapping the partially ordered set of classes of a schema into a graph structure.

By definition a schema S is structurally consistent. Every time an update is performed on a schema S and results in a new schema S', the graph corresponding to S' is analyzed to check whether S' is a schema.

For more details the reader is referred to [DelZi91] where a graph-theoretic approach is used to ensure the O_2 schema structural consistency when performing schema updates.

While most of the work done on schema updates has concentrated on preserving structural consistency [B*87], [PS87], an important and difficult area of investigation is behavioral consistency. In Section 4 we sketch a new data-flow technique to detect behavioral inconsistencies.

4. A DATA FLOW TECHNIQUE TO DETECT BEHAVIORAL INCONSISTENCIES

Our goal is to find a pragmatical workable solution to avoid the risk of behavioral inconsistency due to run-time type errors. It is known that type safeness of a general schema is an undecidable problem [AKW90, HTY89]. However, using a simple data flow analysis technique we can express sufficient conditions ensuring type safeness. If such conditions are not satisfied then a run time error might occur; whether a run time error really occurs or not depends on the actual execution flow of the application using the schema. This research is currently in progress. We present in the sequel our preliminary results.

Examples of run-time type errors are: a wrong input to a method, an incorrect assignment, a reference to a non existing method or class type. We do not consider the case of non-terminating method execution.

Standard compilers are not always able to detect all type errors. Let us consider the following example:

```
Class A
    type tuple (a: Integer)
    method get_a : Integer is public
    body { return (self.a) }
end A

Class B inherits A
    type tuple (b: Integer)
    method get_b : Integer is public
    body { return(self.b)}
end B

Class X
    type tuple (x1 : A)
    method m1(t:A) : Integer is public
    body{return (self.x1.get_a + t.get_a)}
end X

Class Y inherits X
    type tuple (y1 : B)
end Y
```

```
Class Example
        type tuple (x : X; y : Y; p : A)
        method m3 : Integer is public
        body{
1               p= new(A);
2               y= new(Y);
                ...
3               x : = y;
4               return(self.x.m1(p))}
                /* Returns the value obtained by adding
                the value of the attribute x1 of x and the
                value of the attribute a of p */
end Example
```

Suppose we want to redefine the method m1 in class Y as follows:

```
Add method m1(t:B) : Integer in class Y is public
body { return(self.y1.get_b + t.get_b) }
```

Note that the new definition of m1 in class Y satisfies the covariance rule and most compilers (e.g. Eiffel, O2) will accept this change. After the update method m3 will invoke the new method m1 defined at Y since *x* is bound to an object of class Y (because of the assignment at line 3). Hence m1 will try to access the attribute *b* of its actual parameter *p*. But *p* is bound to an object of class A which does not have an attribute named *b* and this causes a run time error to occur.

We sketch in the rest of this section the data flow technique which detects this kind of errors.

4.1 The Data Flow Analysis

We associate to each attribute x_i belonging to a class C_i a set S_i that contains the type of any object that x_i may be bound to at run time. S_i is called the *type set* of x_i. Initially S_i contains the static type of x_i (i.e. the type specified in the declaration of x) and all of its subtypes (if any). In the example, the *type set* S_X of the attribute x in Class Example is {X, Y}.

It is possible to prove that by looking at the *type set* associated to each attribute in the schema it is possible to identify run time type errors.

Let us define the set of all pairs $<x_i,S_i>$ where x_i is an attribute belonging to a class C_i and S_i is its associated *type set*. We call this set the *T-Descriptor* of C_i. To build *type sets* we introduce the function ***exec***(I, TD) which given a statement I, and a T-Descriptor TD returns the T-Descriptor after the execution of I. In what follows we provide the definition of the function ***exec*** for some typical statements of an imperative language:

Sequence of statements $(I_1; I_2;...;I_n)$

$$exec(I_1; I_2;...;I_n, TD) = exec(I_2;...;I_n, exec(I_1, TD))$$

The T-Descriptor is computed recursively by applying **exec** to each statement.

Object dynamic creation (x_i = new(C_i))

exec(x_i= new(C_i), TD) = TD', where

TD: {<x_1,S_1>,...,<x_i,S_i>,...,<x_n,S_n>}, TD': {<x_1,S_1>,...,<x_i,{T_i}>,...,<x_n,S_n>} and T_i is the type of class C_i.

The type set of the variable bounded to the newly created object initially contains just the type of the object created.

In the example in method m3 the T-Descriptor after the analysis of the statement at line 2 is (x retains its initial type-set value {X, Y}):

TD: {<x,{X,Y}>, <y,{Y}>, <p,{A}>}

Assignment statement ($x_i := x_j.m(x_1,...,x_n)$)[1]

exec($x_i := x_j.m(x_1,...,x_n)$, TD) = TD' , where

TD: {<x_1,S_1>,...,<x_i,S_i>,<x_j,{T_{j1},...,T_{jm}}>,...,<x_n,S_n>}

Let T_{j1} (T_{11},...,T_{n1}) → U_1 ,..., T_{jm} (T_{1m},...,T_{nm}) → U_m be the signatures of every occurrence of m defined respectively in T_{j1},...,T_{jm} and suppose all such occurences of m conform to the covariance rule. Thus TD' becomes:

TD': {<x_1,S_1>,...,<x_i,{U_1,...,U_m}>,<x_j,{T_{j1},...,T_{jm}}>,...,<x_n,S_n>}

The variable x_i in the left hand side of the assignment has the same type set of the expression in the right hand side of the assignment.

Referring to the example, in the body of m3 the T-Descriptor after execution of line 3 is:

TD: {<x,{Y}>, <y,{Y}>, <p,{A}>}

Conditional statement (**if** C **then** I_1 **else** I_2 **fi**)

exec(**if** C **then** I_1 **else** I_2 **fi**, TD) = ***exec***(I_1, TD) • ***exec***(I_2, TD), where

{<x_1,S_1>,...,<x_n,S_n>} • {<x_1,U_1>,...,<x_n,U_n>} = {<x_1,$S_1 \cup U_1$>,...,<x_n,$S_n \cup U_n$>}

The type set associated to each variable is the set union of the type sets resulting at the end of both branches of the conditional statement.

For reason of space limitation we omit the definition of the function ***exec*** for the other typical statements of an imperative language. The interested reader is referred to [CLZ91] for such definitions.

Sufficient Conditions

A (run time) type error may occur if one of the following sufficient conditions for type safeness is violated:

1) Type sets contain the types that the associated attribute may be bound to at run-time; therefore, at each step of the data flow analysis, every type set should be totally ordered with respect to the subtyping relation. In addition, the type set should be superior limited by the static type of the associated attribute. Attributes cannot refer to objects belonging to a superclass of their static type. For example, in the class Example the type set of the attribute x should contain only X and/or subtypes of X.

2a) If a method is invoked on an attribute (e.g. x.m) then there must exist a definition of the method, either locally or inherited, for every type belonging to the type set of this attribute (x).

2b) Type sets associated to every actual parameter of a method should be superior limited by the static type of their corresponding formal parameter. As a consequence any object that is passed as a parameter to a method conforms to the definition of the corresponding formal parameter.

In the example, the T-D descriptor for method m3 in class Example during the call of m1 (line 4) is: {<x,{Y}>, <y,{Y}>, <p,{A}>}. Condition 2b is violated at line 4, since the type set of the actual parameter p contains the type A and we have that p is not superior limited by the class B. Therefore we conclude that executing method m3 will cause a run-time type error.

5. OBJECT UPDATES

Object instances have to be modified in accordance to the schema change. There are three basic approaches to perform schema updates with respect to the database:

- Screening,

---

[1] We consider only assignment statement of the form $x_i := x_j.m(x_1,...,x_n)$; this should not be viewed as a restriction since any assignment can be transformed into an equivalent (sequence of) new one(s) having the previous form. For instance x := y; can be transformed into x := y.m, where method m simply returns *self*. The composition of methods such as x := y.m2.m1 can be transformed into the following sequence of assignments: temp := y.m2; x := temp.m1.

- Immediate conversion,

- On demand conversion.

The comparison of these approaches is not simple, as it requires a cost-analysis and simulation techniques. A first preliminary analysis is reported in [Zic91b].

6. CONCLUSIONS

We have designed and implemented a first prototype of a tool which ensures structural consistency when updating an O_2 schema. The tool, called Interactive Consistency Checker (ICC), allows an interactive dialogue with the schema designer. The ICC given a schema and a proposed update, detects whether structural inconsistencies may occur. It then refuses those updates which produce structural inconsistencies. The reason for the refusal of the update is always given to the user. A detailed description of the ICC is reported in [DelZi91]. We have implemented a second version of the ICC tool for the object-oriented database language SOL developed in the Esprit project 2443 "Stretch" [ZCT91].

We are now designing a new tool which is a front-end of the compiler with the following goals:

- Reducing the number of recompilations every time a schema update is performed. We are extending well known techniques for smart recompilation to the object-oriented paradigm.

- Detect possible behavioral inconsistencies due to run-time type errors. The tool will use the data flow technique shown in section 4.

- Helping the schema designer in performing schema updates. To this purpose, it is useful to group updates together: The notion of *transaction updates* allows the transformation of certain illegal updates into a sequence of updates which result in legal schema manipulations.

The most obvious evolution of our approach consists in the integration of the various tools in the compiler itself.

Moreover, we are studying the problem of schema updates in the presence of a set of *constraints* associated both to the schema and to the instance database.

Acknowledgments

Rakesh Agrawal provided useful comments on an earlier version of this paper.

REFERENCES

[AKW90] Abiteboul S., P. Kanellakis, E. Waller, " Method Schemas", in Proc. ACM, PODS 1990.

[B*87] Banerjee et al. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", Proc. ACM SIGMOD, 1987.

[CLZ91] Coen A., Lavazza L., Zicari R., " Updating the Schema of an Object-Oriented Database System", Politecnico di Milano, Research Report, 1991

[DelZi91] DelCourt C. , Zicari R., "The Design of an Integrity Consistency Checker (ICC) for an Object-Oriented Database System", Politecnico di Milano, Report 91-021, April 1991. A short version in Proc. ECOOP, Geneve, July 1991

[HTY89] Hull R., K. Tanaka, M. Yoshikawa "Behavior Analysis of Object-Oriented Databases: Method Structure, Execution Trees and Reachability", Proc. 3rd Intl. Conf. on Foundations of Data Organization and Algorithms, Paris, June 1989.

[LR89] Lecluse C., Richard P., "O2, an Object-Oriented Data Model", Proc. 15th VLDB Conf., Amsterdam, August 1989.

[Mey88] Meyer, B. *"Object Oriented Software Construction"*, Prentice Hall International - Series in Computer Science (1988).

[PS87] Penney D.J., Stein J., "Class Modification in the GemStone Object-Oriented DBMS", ACM OOPSLA, October 1987.

[Pan88] Report on the Object-Oriented Database Workshop: Panel on Schema Evolution and Version Management, SIGMOD RECORD, Vol. 18, No.3, September 1989.

[ZCT91] Zicari R., Ceri S. , Tanca L., "Interoperability between a Rule-based Database Language and an Object-Oriented Database Language", in First Int. Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan, April 7-9, 1991.

[Zic90] Zicari R., "Primitives for Schema Updates in an Object-Oriented System: A Proposal", OODBTG Workshop, X3 IPS, SPARC/DBSSG/OODBTG , October 23, Ottawa, Canada.

[Zic91a] Zicari R., "A Framework for Schema Updates in an Object-Oriented Database System", short version of [Zic91b] in Proc. 7th IEEE Data Engineering Conference, April 8-12, 1991, Kobe, Japan.

[Zic91b] Zicari R., "A Framework for Schema Updates in an Object-Oriented Database System", in "The O2 book", (F. Bancilhon, C. Delobel, P. Kanellakis eds.), Morgan Kaufman, 1991 to appear.

An Overview of Integrity Management in Object-Oriented Databases

Won Kim
Yoon-Joon Lee[1]
Jungyun Seo[1]

UniSQL, Inc.
9380 Research Blvd.
Austin, TX 78759
(512)343-7297

[1]*Department of Computer Science*
KAIST
Daejeon, Korea

1 Introduction

Ideally, a database system should be designed to enforce arbitrary integrity conditions that the user may be able to specify [3, 2]. Although in the mid-seventies, researchers investigated the semantics and implementation architecture of integrity constraints and triggers, current commercial database systems enforce a minimal set of integrity conditions, largely because of the performance overhead that enforcement of integrity conditions incurs. However, during the past several years there has been a perceptible renewed interest in integrity constraints and triggers [7], both in the context of active data management and method support in object-oriented database systems. We feel that a classification of integrity conditions on the basis of their performance impact may prove to be a useful basis for the current research and development efforts. It may provide a good basis for the design and implementation of database systems that can support a significantly richer environment for automatic preservation of user-specified database integrity than is provided by the current database systems.

In this paper, we provide a preliminary discussion of a framework for classifying integrity constraints in the context of an object-oriented data model. An object-oriented data model gives rise to additional types of constraints beyond those meaningful under the relational model. A fuller treatment of the issues of integrity management in object-oriented databases is given in [5].

2 Types of Integrity Constraints

We identify five types of integrity constraints in the context of relational databases on the basis of a qualitative complexity of their evaluation. We note that the classification does not presume the existence of special data structures (e.g., indexes, sorted tables). This classification, with certain extensions, is also applicable to object-oriented databases.

Type-1: A single column within a single record: constraints which can be checked in a single column within a single record.

Type-2: A single record: constraints which can be checked in multiple columns within a single record.

Type-3: A set of records of a table: constraints which require access to a set of records in a table.

Type-4: All records of a single table: constraints which require access to all records of a table.

Type-5: Records of multiple tables: constraints which require access to records of multiple tables.

If we adopt the view that an object-oriented data model is a generalization of the relational data model, the five types of integrity constraint for the relational model have direct counterparts under an object-oriented model. All that is necessary is to substitute the terms record and relation with object and class, respectively.

However, an object-oriented data model includes several concepts that the relation model does not have. We now examine how these additional concepts gives rise to additional constraints in object-oriented databases. The following are the additional concepts supported in object-oriented databases.

- An object encapsulates attribute values and methods.
- An attribute may have an arbitrarily complex neseted object as its value.
- An attribute may have a set of values.
- Classes are organized in a class hierarchy.

Methods

There is no corresponding notion of methods in the relational data model. Methods can be written in any programming language, and, as such, have universal computing power. Since a method is defined for a class, it can be used in specifying constraints on the objects of the class. Therefore, users are not limited to using only query expressions to specify constraints as in relational database systems. However, users must be aware of the complexity of the methods used in constraint specifications and be prudent in the use of methods so as not to degrade system performance due to constraint checking.

Nested Attributes

Unlike the relational model, an object-oriented model allows an attribute to have as its value an instance of an arbitrary class. If an attribute of an object, say `Obj1`, has as value another object, say `Obj2`, users can use attributes of `Obj2` (and the objects that attributes of `Obj2` has as their values, and so on) in the constraint specifications for `Obj1`.

The traversal of a nested attribute is equivalent to a *join* in relational databases. To check a constraint on a nested attribute, the system must access objects in more than one class. Therefore, a constraint specification on a nested attribute is in general type 5 in our classification.[1]

[1] Depending on implementations, the system may not have to check all instances of multiple classes for a constraint on a nested attribute; the check may be limited to a constant number of data accesses.[4]

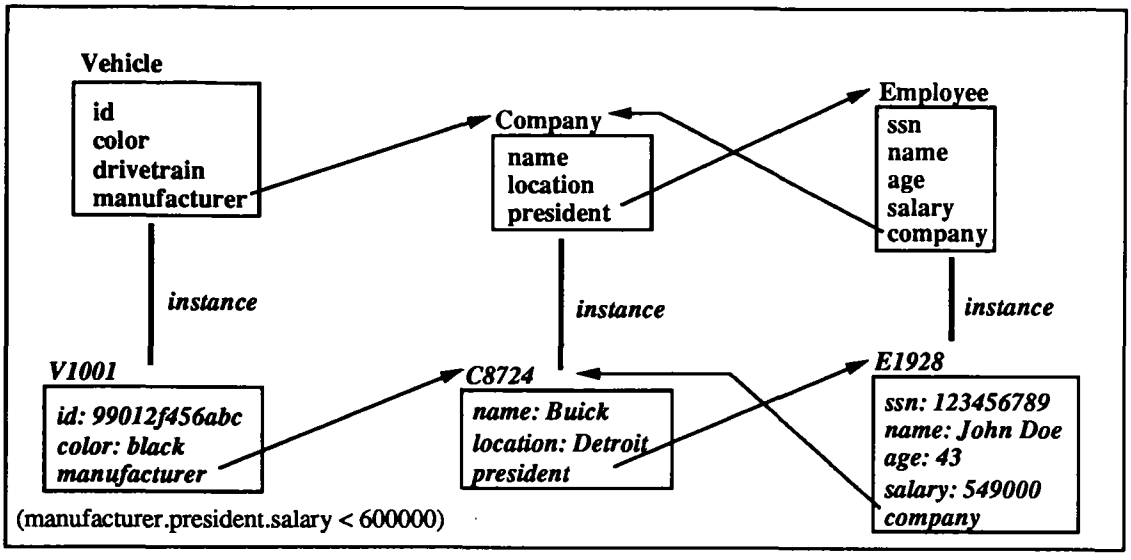


Figure 1: Placement of a constraint on a nested attribute

Another interesting question concerning constraints on nested attributes is whether the constraint logically belongs to the starting class or the terminal class of the nested attribute. For example, let us consider the following constraint specification for the class `Vehicle` in Figure 1 which shows relevant classes in the schema and objects of the classes.

```
( manufacturer.president.salary < 300000 )
```

This constraint involves objects of three different classes: `Vehicle`, `Company`, and `Employee`.

Intuitively, since the constraint is defined in the class `Vehicle` it should be placed in the definition of `Vehicle`, the starting class of the nested attribute. In this case, the constraint is applicable to every instance of `Vehicle`. The constraint will be checked when the system attempts to access the value of an instance of `Vehicle`, say `V1001.manufacturer.president.salary`.

On the other hand, since the actual value which must be compared is in one of the objects of the class `Employee`, the constraint may need to be part of the definition of the class `Employee`, the terminal class of the nested attribute. The justification of this approach is that in object-oriented databases, an object, identified by a unique identifier, is a logical unit of access, and therefore a constraint may be associated with a single object, rather than a class. However, since this constraint is meaningful for the class `Vehicle` and only one instance, `E1928`, of `Employee`, it does not seem appropriate to place the constraint in the class definition of `Employee`.

Set-Valued Attributes

The fact that an attribute may have a set of values from its domain gives rise to a new type of constraint; namely constraints on the cardinality of the set of values for the attribute.

Ex. 2.1 *A new class definition for* `CountryClub`.

```
( CREATE-CLASS CountryClub
    :ATTRIBUTE ( (member :cardinality MAX 370
                        :domain person ) ) )
```

In this example, the maximum number of members of a country club is 370. The constraint may also be specified with `MIN` or `EXACT` to specify the minimum and exact number of elements in a set of values. This type of constraint falls into type 3 in our classification.

Complex Objects

A complex object is in general a cluster of related objects in which an object may recursively reference any number of other objects. A complex object is modeled by combining the concepts of a nested attribute and a set-valued attribute. In general databases for engineering-oriented applications are collections of complex objects, in which an object references other objects and an object may in turn be referenced (shared) by any number of other objects. When an object is updated or deleted, or a new version of the object is created, some or all other objects that reference it may become invalid, and thus need to be notified of the change [6, 1].

It is desirable for users to be able to specify constraints for change notification. For example, a user may need to specify the following constraint for a CAD application: *when memory-layout-101 is updated or deleted, send a change notification to board-design-group1.* We note that this constraint is defined not on the class `memory-layout`, but on one instance (`memory-instance-101`) of the class. This type of constraint falls into type 1 in our classification, since it can be checked against the single object on which the constraint is defined. In relational databases, users can only specify constraints on a table, rather than on a single tuple of a table.

Class Hierarchy

A class hierarchy represents the *isa* relationship between classes. The *isa* relationship means, among other things, that any instance of a class is logically an instance of the superclasses of the class. For example, when a user is to select all instances of `Vehicle`, the system may need to access all (direct) instances of `Vehicle` and all instances of the subclasses of `Vehicle` (indirect instances).

A constraint involving only direct instances of a class is of type 3, since it may be evaluated by accessing only the instances of the class. However, a constraint on all direct and indirect instances of a class requires access to more than one class, and is of type 5 in our classification.

3 Conclusion

In this paper, we discussed integrity constraints to which the paradigm shift from the relational model to an object-oriented model of data gives rise. This is one of the aspects of research into object-oriented database systems which have not received adequate attention up to now. The preliminary results we presented are a part of our broader research into integrity managment in active object-oriented database systems. The broader research is motived by our wish to establish a classification of events, conditions, and actions in triggers in active data management systems which will be useful in helping system developers and users to understand the tradeoff between the performance and flexibility in specifying integrity constraints.

4 Types of Triggering Events and Actions

Due to space limitations, in this section we provide only a brief outline of our classifications of events that trigger constraint checking, and actions that a database system may take upon being triggered. More details are provided in [5].

In the context of database systems, we can categorize events into two classes; *database access event* and *non-database access event* (or external event). The "intuitive" criterion we use for classifying events is the overhead to the system for processing the events. For events involving a small number of objects (e.g., an update of a single record), users will expect fast response; while users will be more tolerable of slower responses for events involving a large number of objects (e.g., a timer interrupt to checkpoint the database). When users can be expected to tolerate slower responses, complex triggering-event specifications may also be tolerable. However, for events which require quick responses, complicated trigger specifications may not be desirable.

In most existing database systems, the only action taken in response to a violation of constraints is to reject the transaction and set a pre-defined parameter with some error code. This is often inadequate, and database systems should allow a set of more useful actions. Actions can roughly be categorized as follows:

- reject transaction
- database updates
- other general events

"when deleting the table `Department`, update `Employee.dept#` to NULL", and "if the object `memory-layout101` is updated, set the update-timestamp to the current time" are examples of triggered database updates. Other general events include any action not related to the database, such as "call system operator", and "send an e-mail to cpu-design-group" are examples of general events not related to the database.

References

[1] H.-T. Chou and W. Kim. A unifying framework for version control in a cad environment. In *Proceedings of the International Conference on Very Large Data Bases*, pages 336–344, 1986.

[2] C. J. Date. *An Introduction to Database Systems.* Addison Wesley, 4 edition, 1987.

[3] E. B. Fernandez, R. C. Summers, and C. Wood, editors. *Database Security and Integrity.* Addison Wesley, 1981.

[4] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[5] W. Kim, Y.-J. Lee, and J.-Y. Seo. Integrity Management in Object-Oriented Databases. UniSQL, Inc. Technical Report, Feb. 1991.

[6] T. Neumann and C. Hornung. Consistency and transactions in cad databases. In *Proceedings of the International Conference on Very Large Data Bases*, Sept. 1982.

[7] M. Stonebraker, E. Hanson, and C.-H. Hong. The design of the postgres rules system. In *Proceedings of International Conference on Data Engineering*, pages 356–374, 1987.

Supporting Views in Object-Oriented Databases

Marc H. Scholl and H.-J. Schek

ETH Zurich, Dept. of Computer Science, Information Systems – Databases
ETH Zentrum, CH–8092 Zurich, Switzerland. E-mail: {scholl,schek}@inf.ethz.ch

1 Introduction

Relational database systems provide a powerful abstraction mechanism: any query, since it returns a relation, can be used to define a *view*, that becomes a derived (or virtual) relation. Views are defined by a statement such as "**define view** <*name*> **as** <*query*>." Views can be used to tailor the global database schema.

- Query formulation is simplified, if frequent subexpressions are predefined.
- Application programs are insulated from changes to the underlying schema.
- Information can be restructured to better suit an application programs' requirements.
- Derived information is kept consistent with base data.
- Access restrictions (for authorization) can be enforced by hiding data.

In contrast to base relations, views are typically not stored permanently, but rather computed on demand. Queries to view relations are modified by query substitution so as to operate on the underlying base relations. Therefore, any update to a base relation propagates automatically to all views defined over it. Conversely, view relations can not be updated freely, since it is often ambiguous how to trace view updates back to updates of base tuples. Typically, only views containing the key of their (one) underlying base relation can be updated.

The view mechanism should also be offered by the next generation, object-oriented DBMSs. Object identity (OId) will alleviate the view update problem. If objects are identified independent of the values associated with them, it is possible to propagate view updates back to base objects.[1] If we want to define views as in relational DBMSs, by queries, this has important consequences on the query language. Most importantly, queries will have to preserve object identity. The other possibilities discussed in the literature are object-generating queries and queries returning data values (e.g., relations). We will restrict ourselves to queries returning *existing* objects. Otherwise, updates to query results would not propagate to the original objects. If queries return original objects, the main questions are: How can we allow restructuring operations (if they change the type of objects)? Is it possible to use query operators similar to the relational projection or join, and still have queries deliver base objects? What flexibility is needed in the type system, and can we still apply (some) static type checking?

These issues are addressed in this brief overview of the work performed in the COCOON project at ETH Zurich [8, 9]. We first discuss *object-preserving query semantics* of a generic object-oriented query language in the style of a relational algebra. Then we show how views defined by such query expressions can be updated: We elaborate on certain fundamental properties of the object model, such as the separation between types and classes, multiple instantiation (an object may be an instance of several types at the same time), and multiple class membership (an object may be a member of several classes at the same time).

2 Terminology

We use the object-function model called COCOON [9], that has been developed as an evolution from and generalization of the nested relational model [7]. The IRIS model [11] with its roots in DAPLEX has similar features. Objects are pure abstractions, all data (or objects) are associated to them (as "state" or "related" objects) by functions.[2] This means that "attributes", "components", or "instance variables" are not distinguished from "derived" or "computed" values. Similarly, we can often neglect the difference between (retrieval) functions and (update) methods, and treat them in the same way. We use the following terminology:

- *Objects* are instances of abstract types, specified by their interface operations. *Data* are instances of concrete types (e.g., numbers, strings) or constructed types, such as tuples or sets [2, 9].

[1] Formally, object identity is a prerequisite for updates: without OId we could only replace, but never modify data.

[2] In implementation terms, this means: the object itself is just an identifier. The "state" of the object is the collection of return values of all functions defined on it (that map the OId to other objects, i.e., OIds , or values).

- *Functions* are either retrieval functions or update methods. They are described by their name and signature (that is, domain and range types). Functions may be set-valued.
- *Types* are described by their name and the set of functions that are applicable to their instances. Types are arranged in a *subtype hierarchy*, where subtypes inherit functions from their supertypes. Objects can be instances of more than one type at the same time ("multiple instantiation").
- *Classes* are typed collections of objects (sometimes also called "type extents"). Classes are arranged in a *subclass hierarchy* that is exactly the set inclusion between the sets of objects they represent. Objects are "members of" classes, possibly more than one at a time ("multiple class membership"). Particularly, superclasses contain all members of their subclasses.

We note that the separation of types and classes is essential. In the relational model, a type corresponds to a relation's schema (the structure), a class to a relation's extent (the set of tuples). At first glance, classes could be viewed as (persistent) sets of objects. However, a set is data, whereas a class is an object [2, 9]. That is, it has an identity that is independent of that set of class members. Each class object is an instance of the type *classtype*, which has (among others) two important functions: For each class *C*, *membertype(C)* returns the associated type, and *extent(C)* returns a set of objects of this type, the class members. Class extents are polymorphic sets: member objects may be instances of many different types. However, they are uniform in that every member is (among others) an instance of *membertype(C)*. In the query language, where we use classes as arguments, type checking is based on this unique member type.

An important additional feature distinguishes our model from others: class predicates. These are usually found in knowledge representation (classification) languages such as KL-ONE [4, 3]. Our classes may be constrained by a predicate that must be satisfied by all members of the class. We distinguish two cases: class predicates may be only necessary or necessary and sufficient conditions. Class predicates can serve several purposes: first, they allow the specification of integrity constraints (necessary predicates). Second, they are our means of separating compile-time type checkable aspects of object types from run-time checks: the former are part of type definitions (e.g., function signatures), the latter are expressed as class predicates (e.g., cardinality restrictions for set-valued functions). Third, and most importantly, class predicates can be used to handle updates to (selection) views: if class predicates are necessary and sufficient conditions, then all (common) members of the superclass(es) that satisfy the predicate are *automatically classified* into the subclass. Conversely, if objects in a class (due to updates) no longer satisfy the class predicate, they are re-classified (recursively) to belong to the superclass only.

As an example, consider two classes *Persons* and *Adults*, both of the same type *persontype*. We know that a person is an adult, if and only if his or her age is over 17. So we define a necessary and sufficient class predicate for class *Adults* that will automatically include members of the *Persons* class into the subclass whenever their age is 18 or more. The system should automatically remove persons from the subclass (and keep them in the superclass only) whenever their age is changed to a value below 18. In COCOON, this situation is represented as follows:

define type *persontype* **isa** *objecttype* = *name*: string, *age*: integer, ...;
define class *Persons*: *persontype* **some** *Objects*;
define class *Adults*: *persontype* **all** *p:Persons* **where** *age(p)* > 17;

Type definitions list the (set of) supertypes (*objecttype*, the predefined top of the type hierarchy in our example) and the applicable functions with their range types. Class definitions include the membertype and the (set of) superclasses (*Objects*, the predefined top of the class hierarchy, for *Persons* and *Persons* for *Adults*). The optional predicate (none is present for the class *Persons*), is a necessary condition in case of a **some** qualifier, and necessary and sufficient in case of an **all** qualifier. Therefore, in all valid database states, the extent of class *Adults* will always be exactly those members of class *Persons* for which the *age* function returns a value over 17. Notice, that this class definition of *Adults* is just what we would expect from a (selection) view defined over the *Persons* class.

3 Object-Preserving Query Semantics

We use a *set-oriented* query language similar to relational algebra, where the inputs and outputs of the operations are *sets* of objects. Hence, query operators can be applied to extents of classes, set-valued function results, and query results. Many such object-oriented algebras have been proposed in the literature. We can distinguish three approaches to the exact semantics of queries, depending on what the result of queries are:

1. "*Relational semantics*": query results are data values, not objects. For example, every query may return a set of tuples containing some values describing properties of objects. This semantics is useful for generating query outputs (we do not want to deliver objects, or OIds, to the user), but it is not suited for the definition of (updatable) views, since object identity is lost, so updates make no sense.
2. "*Object-generating semantics*": queries generate new objects. The states of the new objects are (partial) copies of the states of qualifying objects. Again the problem is how to propagate updates back to the original objects. This kind of query semantics is motivated by object models that do not allow objects to be instances of more than one type (class). If the type ("structure") of objects is modified by the query (e.g., a projection), the result has to be a set of new objects.
3. "*Object-preserving semantics*": queries return (some of) the input objects. As an immediate consequence of type-changing query operators, such as projections or "joins", we have to allow multiple instantiation. Multiple class membership is a consequence of making query results classes. This semantics for queries allows the application of methods and generic update operations to results of a query, since these contain base objects.

To define updatable views by means of queries, we should opt for object-preserving operator semantics. Otherwise, one must play some implementation tricks when voting for other query semantics in order to provide updatable views. For example, one can internally keep the original OIds together with the new objects. However, object-preserving query semantics is the cleaner concept. Then views are additional (virtual) classes that need to be positioned in the class hierarchy, and their membertypes need to be positioned in the type hierarchy. The objects in these view classes are base objects.

In the sequel, we give a brief overview of the COOL query language and its semantics in terms of result types and extents. In COOL, operands are sets of objects, the operators are the relational algebra operators with appropriate extensions (syntactically, operands may be classes: formally, the operands are the *extents* of the classes). Query results are also sets of objects. View definitions introduce new (virtual) classes, whose extent is defined by the query. For views defined by each of the basic COOL operators, we describe what the membertype and extent is, and how these are positioned in the type and class hierarchies. For ease of presentation, assume that all views are defined over base classes. In general, views may also be defined over other views, or by composite queries (see [8] for a detailed exposition). In the following, let C be a class with member type T.

Selection (**define view** V **as select** $[P]$ (C)). The view class V is a subclass of the base class C, with the same member type T. We now have two classes, V and C, of type T. The extent is the subset of C-members satisfying P. In fact, the effect of the view definition is precisely the same as if class V were defined with a necessary and sufficient class predicate: "**define class** V:T **all** C **where** P".

Projection (**define view** V **as project** $[f_1, ..., f_n]$ (C)). The view class V is a superclass of C. The member type, say T', of V is a supertype of T (less functions are defined, only those listed in the projection: $f_1, ..., f_n$), the extent of V is that of C. The effect is the same as a schema definition containing a statement "**define class** V: T' **all** C **where true**".

Extend (**define view** V **as extend** $[f_i$:=<$expr_i$>,...] (C)). Projection eliminates functions, extend defines new derived ones. <$expr_i$> can be any legal arithmetic, boolean, or set-expression. The view V is a subclass of C: their extents are the same and the member type of V, say T', is a subtype of T (it has the old functions plus the new ones). The effect is the same as "**define class** V: T' **all** C **where true**".

Set operations. As the extent of classes are sets of objects, we can perform set operations as usual. Their effects on class extents is their standard set theoretic semantics. Due to the polymorphic type system, we need no restrictions on the operands' member types (ultimately, all objects are instances of *objecttype*). The member type of the result, however, depends on the operands' types: A union view is a common superclass of the base classes, whose member type is the lowest common supertype (in the type lattice) of the input types. Difference views are subclasses of their base class with the same membertype; finally, an intersection view is a common subclass with a member type that is the greatest common subtype of the input types.

4 View Updates

Beyond *type-specific* update operations (i.e., methods), we provide a collection of *generic* update operators to facilitate set-oriented processing. First, there is a set-iterator for updates, **update** [m] (<*set-expr*>), that take as

arguments a set of objects and an update operation, *m*, to be performed on all elements. The other generic update operators are **insert** and **delete** for creating and destroying objects, **add** and **remove** for including and excluding existing objects into/from sets (in contrast to **insert** and **delete**, **add** and **remove** have no effect on the existence of the objects), and **set** to assign values (data or objects) to functions.

For type-specific update operations (methods), there are no restrictions whatsoever on view classes and all methods included in the view's membertype can be invoked. Notice, that type-changing operations, such as projections, deal with update methods and retrieval functions in the same way: a projection list includes the methods that shall be visible in the view! For the generic update operators of COOL, the semantics of their application to view classes is always defined to be exactly the same as if the view class were defined as a base class with a necessary and sufficient class predicate ("**define class** *V*...**all**...**where**..."). Therefore, the foundation of our update semantics is automatic classification: updating objects may cause the objects to dynamically change class memberships. The alternative solution of disallowing all object modifications that cause class predicates to change their truth value is too restrictive.

Consider the following scenario: Let a class *Persons* have a subclass *Myfriends*. Certainly, we can not express a sufficient predicate on persons to decide who are my friends. So, we need to tell the system explicitly, by the **add** operation, which persons to put into that subclass. Suppose there is another subclass, *NewYorkers*. Obviously, this subclass can be defined with a (necessary and) sufficient predicate, namely "**define class** *NewYorkers: personype* **all** *Persons* **where** *addr*='NewYork'". Alternatively, we could define *NewYorkers* as a view by the statement "**define view** *NewYorkers* **as select** [*addr* = 'NewYork'](*Persons*)". In any case, we expect from the system to (i) automatically add a *Persons* member to *NewYorkers*, if the *addr* function is set to New York, and (ii) to remove an object from *NewYorkers*, if *addr* is set to some other value. The latter is also true if we apply the update to class *NewYorkers*. On the other hand, suppose we add some person object *p* to *NewYorkers* explicitly (by using the generic **add** operator). If the person fails to have a New York address, the update will not succeed, because the necessary class predicate is not fulfilled.

In general, due to the dynamic reclassification of objects based on class predicates, we need to apply only few restrictions to view updates, since most "exceptions" are detected during the evaluation of class predicates. An example for updates that are disallowed is the assignment to derived functions (e.g., in extend-views), the insertion/addition into union views (that could only be allowed if the two base classes have a discriminating predicate, because otherwise we can not disambiguate the insertion), or—the symmetric case—removals from intersection views. Details for each kind of views and update operations are discussed in [8].

We did not mention join views yet. COOL has no join operator. We can express the same semantics by the **extend** operator. Instead of joining two classes, we extend one of them by a new function. The other one may automatically be extended by the inverse function. The new function returns, for each member of that class, the set of "join partners" from the other class [9]. The derived function is defined using a selection applied to the other class, where the predicate depends on the current member of the first class. For example, a view over *Persons* that shows for each person the *neighbors*, that is, those persons living at the same address is defined as

define view *NeighborPersons* **as extend** [*neighbors*:=**select**[*addr(n)* = *addr(p)*](*n:Persons*)](*p:Persons*).

This way of expressing joins is object-preserving, *NeighborPersons* contains base objects, namely all members of *Persons*. Therefore, we can also update "join views": we can modify all information about the person objects and their neighbors, the only restriction is that we disallow setting the *neighbor* function to a new value. We can, however, change its value indirectly by modifying persons' addresses.

5 Fundamental Properties

The following properties of the query language have been essential for the view definition capability and the view update semantics. If some of these properties are not met by a language, our solutions will fail, partly or completely. Thus, the results we obtained are not bound to the COOL language, but to these properties.

Object preservation is the central concept. It is crucial for a view definition facility. Object preserving operator semantics means that the results of queries are *existing* objects from the database, in contrast to object-generating (results are *newly generated* objects) or tuple-generating (results are *data*, not objects) semantics.

The *type/class separation* is a consequence of object preservation: if both projections and selections are to preserve objects, and if composite select/project queries are permitted, we need this separation in order to connect the view class properly with the base class. The position of query results in the type and class hierarchies have to be less precise without this distinction (see [6], where all query results are direct subclasses of "OBJECT"). Furthermore, no operation changes both, type and extent, except for union and intersection of two classes with differing types. So, the separation is a clarification of distinct concepts.

Multiple instantiation and multiple class membership are other consequences of object preservation: since we have type-changing operators (project, extend) all objects in their results "acquired" a new type. If we consider objects in results of queries as being members of the result class as well as the input class(es), we can treat updates to query results in the same way as updates to stored classes and the updates propagate automatically.

Dynamic reclassification during updates: Automatic classification functionality known from AI systems becomes necessary when we take into account, that objects can dynamically gain and loose types during their life time and that changes of an existing object can make it a member of a more specific class (because now it satisfies it's class predicate) or a more general one (if the class predicate of it's current class is violated by the update). Reclassification is the central concept in our view update semantics. While classification (predicate subsumption) is undecidable in general, we try to identify tractable predicates. Furthermore, *re*classification—relative to an original class and a specific update operation—is simpler than classification in general.

6 Related Work

Recently, there have been other proposals for view support in ooDBMSs [1, 5, 10]. These are different from our approach in that we use the standard way of defining views by nothing else than query language expressions. They either introduce special view definition features that duplicate parts of the query language capabilities [1] or use other facilities of their systems. FUGUE [5] uses type hierarchies for information hiding: the user can implement a new type for the view that uses some base type(s) and offers only a restricted functionality or extends the functionality. Also, not all instances of the base type may be exported. POSTGRES [10] uses the rule system to simulate views. Derived tables (views) can be defined by rules and other rules may define specialized update semantics for them.

Acknowledgement. COOL's view mechanism was developed together with Christian Laasch and Markus Tresch. We thank Bharat Bhargava for critical comments on a draft.

References

[1] S. Abiteboul and A. Bonner. Objects and views. Manuscript, INRIA, Dec. 1990.

[2] C. Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5:353–382, Oct. 1990.

[3] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. CLASSIC: A structural data model for objects. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 58–67, Portland, OR, June 1989.

[4] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[5] S. Heiler and S. Zdonik. Object views: Extending the vision. In *Proc. IEEE Data Engineering Conf.*, pages 86–93, Los Angeles, Feb. 1990.

[6] W. Kim. A model of queries for object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 423–432, Amsterdam, Aug. 1989.

[7] H.-J. Schek and M. H. Scholl. The two roles of nested relations in the DASDBS project. In S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*. LNCS 361, Springer Verlag, Heidelberg, 1989.

[8] M. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. Technical Report 150, ETH Zürich, Dept. of Computer Science, Dec. 1990. Submitted for publication.

[9] M. Scholl and H.-J. Schek. A relational object model. In S. Abiteboul and P. Kanellakis, editors, *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, pages 89–105, Paris, Dec. 1990. LNCS 470, Springer Verlag, Heidelberg.

[10] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In H. Garcia-Molina and H. Jagadish, editors, *Proc. ACM SIGMOD Conf. on Management of Data*, pages 281–290, Atlantic City, May 1990. ACM, New York.

[11] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, Mar. 1990. Special Issue on Prototype Systems.

Algebraic Query Processing in EXTRA/EXCESS

Scott L. Vandenberg
David J. DeWitt

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

1. Introduction

The EXTRA/EXCESS DBMS [Care88] consists of an advanced data model (EXTRA) and a non-procedural query language (EXCESS), each of which is described briefly in Section 2. The system is implemented using the EXODUS extensible database system toolkit [Care90]. Motivations for EXTRA/EXCESS, as well as detailed comparisons of it with similar systems, can be found in [Care88]. The goal of this paper is to describe the algebraic fundamentals underlying the processing and optimization of EXCESS queries. The relationship of these fundamentals to other work is detailed in [Vand91]. Here we concentrate on describing the algebraic structures and their operators (Section 3), results on the algebra's expressive power (Section 4), and algebraic query optimization (Section 5). Section 6 summarizes the current status of this work. Many of these results are covered more completely in [Vand91].

2. The EXTRA Data Model and EXCESS Query Language

Two concepts are central to the design of EXTRA/EXCESS: extensibility and support for complex structures with optional identity. In addition, the model incorporates the basic themes common to most semantic data models. Extensibility in EXTRA/EXCESS is provided through both an abstract data type mechanism, where new types can be written in the E persistent programming language [Care90] and then registered with the system, and through support for user-defined functions and procedures that are written in the EXCESS query language to operate on (user-defined) EXTRA types. The EXTRA data model includes support for complex structures with shared subobjects, a novel mix of object- and value-oriented semantics for data, a multiple inheritance hierarchy for tuple types, and support for persistent structures of any type definable in the EXTRA type system (i.e., a strict type-instance dichotomy

```
define type Person:
(
        ssnum:      int4,
        name:       char[ ],
        birthday:   Date
)

define type Employee:
(
        dept:       ref Department,
        salary:     int4,
        kids:       { Person }
) inherits Person

define type Student:
(
        dept:       ref Department,
        advisor:    char[ ]
) inherits Person

define type Department:
(
        name:       char[ ],
        floor:      int4
)

create Employees:   { own ref Employee }
create Students:    { own ref Student }
create TopTen:      array [1..10] of ref Employee
```

Figure 1: A simple EXTRA database

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by an IBM Graduate Fellowship, and by a donation from Texas Instruments.

exists). Figure 1 shows a simple database defined using EXTRA. In EXTRA, the tuple, multiset, and array constructors for complex objects are denoted by parentheses, curly braces, and square brackets, respectively. Object identity is denoted by the **ref** keyword, and the **own** keyword indicates object ownership [Care88]. In EXTRA, subordinate entities are treated as values (as in nested relational models [Sche86]), not as objects with their own separate identity, unless prefaced by **ref** in a type definition or an object creation statement. The declaration **ref x** indicates that **x** is an object identifier (OID).

Briefly, Figure 1 defines four types, all of which happen to be tuple types. The Student and Employee types are subtypes of Person. The semantics of this inheritance are that all attributes and methods of Person are also attributes and methods of Student and Employee, and that all Students and Employees are also Persons (substitutability). Any inherited method can be overridden with a new method body. Figure 1 creates a university database that owns three named (i.e., top-level), persistent objects: Students (a set of Student objects), Employees (a set of Employee objects), and TopTen, a fixed-length array of references to Employees.

The EXCESS query language provides facilities for querying and updating complex structures, and as mentioned above it can be extended through the use of ADT functions and operators (written in E) and procedures and functions for manipulating EXTRA schema types (written in EXCESS). EXCESS queries range over structures created using the create statement. EXCESS is designed to provide a uniform query interface to multisets, arrays, tuples, and single objects, all of which can be composed and nested arbitrarily in EXTRA. The language thus allows for the retrieval, combination, and dismantling of any structure definable in EXTRA. The user-defined functions (written both in E and in EXCESS) and aggregate functions (written in E) are supported in a clean and consistent way.

As an example, the following query finds the names of the children of all employees who work for a department on the second floor; additional examples appear in Sections 3 and 5.

```
range of E is Employees
retrieve (C.name) from C in E.kids where E.dept.floor = 2
```

3. The EXCESS Algebra

EXCESS queries are processed algebraically — the algebraic paradigm historically has proved useful for optimization, models of execution, and theoretical results. The EXCESS algebra [Vand90] provides direct support for many of the advanced constructs of EXTRA/EXCESS, notably arrays, multisets, grouping, inheritance, object identity, and orthogonality of type constructors. The algebra consists of *structures* and *operators*. A structure is an ordered pair (S, I), where S is a *schema* and I is an *instance*. Schemas are digraphs whose nodes represent type constructors and whose edges represent a "component-of" relationship. That is, an edge from A to B signifies that B is a component of A. Each node is labelled with either "set" (short for multiset), "tup" (for tuple types), "arr" (for arrays), "ref" (for object identity), or "val" (an atomic value with no associated structure). These correspond to the four type constructors plus simple values. As in EXTRA, these constructors can be composed arbitrarily — they are completely orthogonal. We also associate a unique type name with each node. Components (fields) of tuples are also named. Every schema has a distinguished root node. "Tup" nodes may have any number of components, "val" nodes have no components, and the other type constructors have exactly one component. Cycles in the graph must contain a node of type "ref". Instances of a schema are elements of the appropriate domain, and the domain associated with a node of a schema graph has been defined (formally; see [Vand91]) to account for object identity and multiple inheritance.

The orthogonal nature of the type constructors of EXCESS (and those of the algebra) has been incorporated into the operator definitions. The algebra is many-sorted, so instead of having all operators defined on "sets of entities" (as in most algebras), we have (for each "sort", or type constructor) a collection of operators that apply *only* to structures whose outermost type constructor is that sort. In particular, the algebra is not set-oriented. For each constructor we introduce a collection of primitive operators that together allow for arbitrary restructurings involving one or two structures of that sort. We list the operators here, but do not describe them in detail; some of them will be more fully defined in the next section (also see [Vand90]). There are eight primitive operators for **multisets** (−, $\uplus$, $\times$, DE, GRP, SET_COLLAPSE, SET_APPLY, SET); three for **tuples** (TUP_CAT, TUP_EXTRACT, TUP); two for **references** (DEREF, REF); and nine for **arrays** (SUBARR, ARR_CAT, ARR_EXTRACT, ARR_COLLAPSE, ARR_DE, ARR_CROSS, ARR_DIFF, ARR_APPLY, ARR). We emphasize that these operators do not depend at all on the schemas of their inputs' components (except, of course, that both inputs of a binary operator must be of the same type, modulo inheritance rules). For example, the unary multiset operators work on any multiset, no matter what it contains. Each operator accepts only inputs of the sort with which it is associated, with the exceptions of the SET, TUP, ARR, and REF operators, which accept any structure as input and place their input inside the

corresponding type constructor (e.g., SET returns a singleton multiset containing its input).

Since algebras are functional languages, we treat predicates in a functional manner. That is, a predicate is an operation (called COMP) that returns its (unmodified) input exactly when the predicate is satisfied (true). Otherwise COMP returns nothing (see [Vand90] for a discussion of nulls in EXTRA/EXCESS). COMP, whose input can be of any type, is the only operator that has a predicate parameter. Isolating predicates within this operator simplifies the other operator definitions as well as their implementations.

We now present a simple query (see Figure 2) to illustrate the flavor of the algebra; more complicated queries appear in Section 5. The query in Figure 2 returns the name and salary of the 5th element of the TopTen array (defined in Figure 1). This query uses the following algebra operators: ARR_EXTRACT, which returns a single, specified element of an array (the result is the element itself, not an array containing the element); DEREF, which, given an OID, returns the value referred to by the OID; and π, which returns a tuple containing only the specified fields of its input (π operates on a single tuple, not on a set of tuples).

Clearly, the operator definitions are heavily influenced by the structures of EXTRA objects. The operator definitions are also motivated by the fact that a rich set of primitive operators enables the derivation of an equally rich set of transformation rules for use in optimization. Furthermore, primitive operators are easier to implement than are complex ones. It is also easy to define non-primitive operators in terms of the primitives. For example, a relational-like join can be defined as follows:

$$\text{rel\_join}_{\Theta}(A, B) = \text{SET\_APPLY}_{\text{COMP}_{\Theta}}(\text{SET\_APPLY}_{\text{TUP\_CAT(field1, field2)}}(A \times B)),$$

where Θ is the join predicate and "field1" and "field2" are the names of the two fields of the Cartesian product.

4. Expressive Power of the Algebra

The algebra was designed to implement the EXCESS query language, not to reflect a database-style calculus such as the relational calculus. Thus the interesting question of expressive equivalence for this algebra is not whether it can express the queries of some formal calculus but whether it can express exactly the queries of EXCESS. This is one motivation for not forcing the algebra to be set-oriented. Naturally, it is crucial that any EXCESS query be expressible in the algebra. The other direction of the equipollence is interesting in that it restricts the optimization alternatives to the smallest set possible given the power of EXCESS and the structure of the algebra and its rules. It also ensures that intermediate steps in the optimization process are always correct representations of EXCESS queries and that any expressiveness results regarding the algebra also apply to EXCESS. We only sketch the proof here; more details are available in [Vand91] and in a forthcoming thesis.

Theorem: The EXCESS query language and algebra are equipollent.

Sketch of Proof: *Reduction of EXCESS to algebra*: The proof that EXCESS is reducible to the algebra is essentially an algorithm that translates any EXCESS query to an algebraic query tree. The proof is an inductive proof that follows the structure of the algorithm (the induction is on the number of certain EXCESS constructs appearing in a query). *Reduction of algebra to EXCESS*: The other direction of the proof is a case-based inductive proof. The induction is on the number of operators in an algebraic expression. An expression in the algebra consists of one or more named, top-level database objects and 0 or more operators. □

A few general remarks about the algebra's power are in order. First, it is capable of simulating most of the algebras mentioned in the literature as long as these algebras do not contain the powerset operator. We conjecture that our algebra is incapable of expressing the powerset, but we have not proved this yet. Such a result would provide an important upper bound on the algebra's expressiveness and computational complexity. This is because the powerset operator, which returns the set of all subsets of its input set, is inherently exponential in nature and (in

---

```
retrieve (TopTen[5].name, TopTen[5].salary)
```

$$\pi_{\text{name,salary}}(\text{DEREF}(\text{ARR\_EXTRACT}_5(\text{TopTen})))$$

Figure 2: A Simple Query

many algebras) it allows for the formulation of least fixpoint queries [Gyss88]. Second, it has been observed that the addition of the powerset operator to some algebras has the same effect as adding while-loops with arbitrary conditions [Gyss88]. Such loops are fundamentally different from the style of loop provided by the SET_APPLY operator. The latter style of loop executes a statement on each element of a (multi)set in turn. The former kind of loop executes a statement many times, but is not restricted to operating on successive elements of a set. Finally, the EXCESS algebra contains some constructs that are second-order in nature. Since the semantics of the algebra are defined operationally (rather than proof-theoretically, as in many calculi), it is not clear that the drawbacks of second-order logics (i.e. incompleteness) have corresponding drawbacks in an algebraic setting.

5. Algebraic Query Optimization in EXCESS

EXTRA/EXCESS queries are optimized by a rule-based optimizer which uses algebraic transformation rules to rearrange a query [Vand91, Grae87]. It determines the cost of each rearrangement using cost functions and statistics and chooses the cheapest one for execution. This section describes a few of the new transformation rules that can be used to optimize EXCESS queries and illustrates them via an example. The algebra is capable of simulating nearly all of the transformations found in the literature. The example presents an EXCESS query over the database of Figure 1 and a series of algebraic representations of that query. None of these query trees is necessarily intended to be the final or optimal plan for the query. Each one represents an alternative execution strategy to be examined by the optimizer. In the example we take some liberties with the details of the algebra in order to clarify the presentation, but we lose none of the essence of the queries. We also use a graphical notation to represent the queries — data flows upward in these graphs, following the arrows. A complete list of new transformations would contain several dozen rules, each of which can be proved sound using the operator definitions. More rules can be found in [Vand91].

The example query retrieves, without duplicates, the names of all advisors of Students, grouped by their students' departments. It demonstrates the use of both object-based accesses (use of the Student.dept reference field) and value-based accesses (the relational-like join). The EXCESS query is:

```
range of S is Students, E is Employees
retrieve unique (S.dept.name, E.name) by S.dept
      where S.advisor = E.name
```

Figure 3 is one way to execute the query — it is similar to what would be produced as an initial query tree by the EXCESS parser and translator. In the query plans shown we make use of the following algebra operators: SET_APPLY, which applies an algebraic expression to all of the occurrences in a multiset; DE, which eliminates duplicates from a multiset; GRP, which groups the occurrences in a multiset into equivalence classes based on the result of an algebraic expression applied to those occurrences; π, which is similar to relational projection but operates on a single tuple[1] ; and "rel_join", which is essentially a relational join. "Rel_join" is not a primitive in the algebra, but is one style of join we have defined using the primitives; see Section 3. Subscripts to an operator indicate parameters to that operator, not its input(s). The plan in Figure 3 joins the two sets using "rel_join", then groups the result (producing a multiset of multisets), performs the final projection, and eliminates duplicates. The final projection actually includes an object-based access (using the algebra's DEREF operator to get to a Student's Department tuple), but we have omitted this for brevity. Note that to process sets nested within sets, the parameter to SET_APPLY can itself contain a SET_APPLY, as in Figure 3. Figure 4 shows the application of a rule pushing DE before GRP:

$$\text{SET\_APPLY}_{DE}(\text{GRP}_{E}(A)) = \text{GRP}_{E}(\text{DE}(A))$$

This is especially advantageous when the duplication factor is large, as it is likely to be here. We simultaneously take advantage of the ability to move π ahead of GRP if the π produces the attributes used by GRP. In Figure 5 we optimize the query further by pushing the DE and π past the "rel_join" node, making use of this rule (among others):

$$\text{DE}(A \times B) = \text{DE}(A) \times \text{DE}(B)$$

This results in DE operating on $|S|$ and on $|E|$ occurrences rather than on $|S| * |E|$ occurrences. The DE and π have been separated into two nodes in Figure 5 to clarify the presentation.

[1] The π used here is easily defined in terms of the three primitive tuple operators listed earlier.

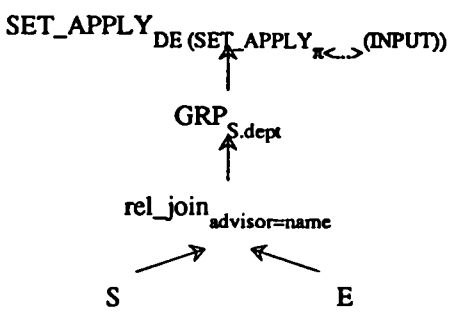


Figure 3: Initial Query

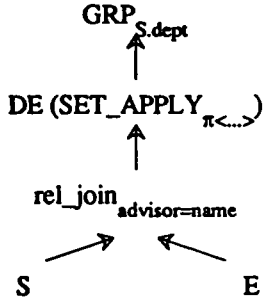


Figure 4: 1st Transformation

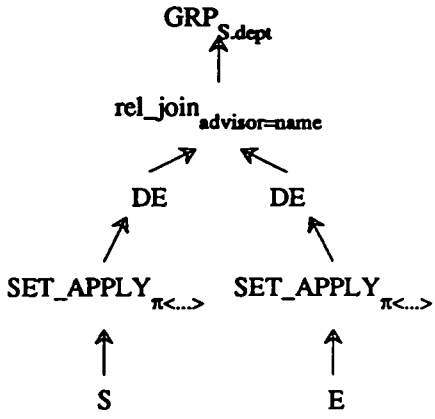


Figure 5: 2nd Transformation

The algebra also provides for compile-time optimization of overridden methods. This particular technique, described in [Vand91], can be used to avoid run-time type checking in some queries.

6. Conclusions, Status, and Future Work

The algebraic paradigm for query processing, as shown here, can be used to successfully model, implement, and optimize many aspects of advanced data models. More complete results may be found in [Vand90, Vand91]. As mentioned in Section 4, the algebra can simulate most other database algebras found in the literature, as long as these do not include recursive capabilities. Any operator of such an algebra can be simulated using an EXCESS algebra expression, and this is true for both object- and value-based algebras, due to the orthogonal introduction of object identity into the EXCESS algebra. This implies that the optimizations used in EXCESS can also be applied to these other algebras and that some systems without algebras can benefit from using the EXCESS algebra.

Much of the system is now operational, including the parser, many of the algebraic operators, the runtime query execution system, the DML support, and support code for the EXODUS optimizer generator, which is being used to build the optimizer. Many of the algebraic rules are present in the current generated optimizer. Some queries have been executed at low levels of the system, and the full system is expected to begin running shortly. Interesting areas requiring future research include normal forms for complex objects, alternative approaches to incomplete information in a complex object database, and the development of more sophisticated statistics for use by the optimizer.

REFERENCES

[Care88] M. Carey, D. DeWitt, and S. Vandenberg, "A Data Model and Query Language for EXODUS", *ACM SIGMOD88 Conference Proceedings*, Chicago, IL, June 1988.

[Care90] M. Carey et al., "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Database Systems*, ed. S. Zdonik and D. Maier, Morgan Kaufmann, Palo Alto, CA, 1989.

[Grae87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *ACM SIGMOD87 Conference Proceedings*, San Francisco, CA, May 1987.

[Gyss88] M. Gyssens and D. Van Gucht, "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra," *Proc. SIGMOD Conf.*, Chicago, Illinois, June 1988.

[Sche86] H. Schek and M. Scholl, "The Relational Model with Relation-Valued Attributes," *Info. Sys.* 11(2), 1986.

[Vand90] S. Vandenberg and D. DeWitt, "An Algebra for Complex Objects with Arrays and Identity", Univ. of Wisconsin-Madison Computer Sciences Technical Report #918, March 1990.

[Vand91] S. Vandenberg and D. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance", *ACM SIGMOD91 Conference Proceedings*, Denver, CO, May 1991.

ENCORE: An Object-Oriented Approach to Database Modelling and Querying

Stanley B. Zdonik and Gail Mitchell[†]
Department of Computer Science
Brown University
Providence, R.I. 02912
sbz,gms@cs.brown.edu

1 Introduction

The ENCORE[1] [7] object-oriented database system has been used at Brown University for a number of years as a vehicle for experimentation in many aspects of next-generation database technology. It was designed as a medium for combining database and programming language technology. It was also designed as an extensible database in which the basic extension mechanism is the abstract data type. As a result, extensions are programmed in a way that conforms to good software engineering practice.

The ENCORE DDL is, as in most conventional database systems, a means for describing data that will be kept in a persistent, shareable store. It can be thought of as a type system. The DML is defined by the ability to invoke methods of abstract types on objects. Thus, ENCORE is a means for creating and manipulating persistent instances of abstract types. A collection of ENCORE type definitions is analogous to a database schema.

The ENCORE type system is a language-independent data model. The implementation of any given type can, in theory, be provided in any programming language. In fact, it is our intention to allow a type T_1 to be implemented in programming language P_1 and a type T_2 to be implemented in programming language P_2. Instances of T_1 and T_2 can be freely mixed in a given ENCORE database. This is possible because the only things that are passed between methods are atomic values and system defined object identifiers (oid's) which serve as handles for ENCORE objects. If a method receives an oid for an object, the only thing that it can do to it is invoke a legal method for objects of that type. For atomic types, we provide translation functions that convert between two different representations (say, a C integer and a LISP integer representation). This does not add much programming overhead because there are only a small number of atomic types. We also disallow the transmission of object representations outside of ENCORE since any safety criteria (e.g., constraints or type checking) that are defined within ENCORE may not be consistently enforced on data while it is outside of the system.

This approach retains the impedance mismatch of conventional databases, but it easily accommodates the implementation of types by multiple languages. ENCORE provides one type system, while each method implementation language provides another. Unlike most current object-oriented database systems, ENCORE is not a complete database programming language. Notice, however, that it would be possible to wrap a programming language around the ENCORE type system (something we are planning to do). The impedance mismatch problem would disappear for types whose methods are implemented in this language.

2 The ENCORE Data Model

ENCORE is based strongly on abstract data types [4]. All types are defined by their interface which is specified in terms of a set of method signatures. A signature for method M is a name plus an ordered list

[†]Support for this research was provided by IBM under contract No. 559716, by DEC under award No. DEC686, by ONR under contract N0014-88-K-0406, by ONR and DARPA under contract N00014-83-K-0146 and ARPA Order No. 4786, by Apple Computer, Inc., and by Texas Instruments.

[1]Many years ago ENCORE stood for Extensible and Natural Common Object REsource. These days it stands for itself.

of types that correspond to the legal types for the arguments of M. Objects can be related to each other by means of these methods. The ENCORE type system has been designed to allow for static type checking. Type equivalence is determined by name equivalence.

ENCORE defines a set of atomic types, **Integer**, **Real**, **Bool**, and **String**. These types define the only values in the system. A value is something which is guaranteed to be immutable. Unlike other object-oriented database systems (e.g. O_2[3]), ENCORE does not allow users to define new value types. All user-defined types describe objects.

A user-defined abstract type is a tuple (S,I) in which S is a specification and I is an implementation. The specification S is a pair (N,M) in which N is a name (i.e., a string) and M is a set of method signatures. The implementation I of an abstract type is a pair (R,P) in which R is a representation type, and P is a set of programs that implement the methods in M in terms of the representation type R. The type R can be any type including an atomic type, an instance of a parameterized type, or any other abstract type. Users of an abstract type are forbidden from knowing the implementation I.

A parameterized type is a mechanism for specifying a family of related types with one textual definition. The parameters for a parameterized type may include other types. In order to retain our ability to do static type checking, we restrict the parameters to be expressions which are statically typed. A parameterized type is like a metatype in that it is a generator for other types. An instance of a parameterized type is a type. For example, **Set[Integer]** is an instance of **Set[T: Type]**.

ENCORE defines two very important parameterized types, **Set[T]** and **Tuple[a_1:T_1, ... ,a_n:T_n]**. These two types play a crucial role in the query facility, where they are used to construct types for query results. When the parameterized type **Set[T]** is given a value such as **Integer** for the parameter **T**, it generates a new type **Set[Integer]**. The name of this type is "Set[Integer]". The type **Set[Integer]** has instances that are sets whose members are constrained to be integers. There can be multiple instances of a type **Set** containing exactly the same members.

A parameterized type **P** defines a set of methods, each of which can be parameterized by the type parameter(s) of **P**. For example, the **Set[T]** type defines an operation with a parameterized signature Insert(S: **Set[T]**,x: **T**). When **Set[T]** is supplied with a value for parameter **T**, the Insert operation is also parameterized with the same value; i.e. type **Set[Integer]** has method Insert(S: **Set[Integer]**,x: **Integer**). The methods defined by a parameterized type do not apply to instances of that type. Instead, when a parameterized type is given a parameter, the newly generated type **N** is given parameterized versions of the methods, which now apply to the instances of **N**. For example, Insert does not apply to the type **Set[Integer]** but applies to instances of that type.

The basic type constructor in ENCORE is the abstract data type, in that it takes a type for its representation (concrete type) and generates a new type as the abstract type. The parameterized types also provide a more standard type construction mechanism.

ENCORE abstract data types are related to each other through subtyping. A type may have many subtypes and many supertypes. Subtyping requires that instances of a subtype can be substituted as instances of any of its supertypes (substitutability). The subtyping mechanism is designed to support strong static typing when the ENCORE type system is embedded in a compiled programming language.

The collection of all current instances of an ENCORE type is the extent of the type. Extents as well as arbitrary sets of instances can be maintained by ENCORE. Sets can be explicitly created or defined by a predicate over some existing set. For example, if set Cars has type **Set[Car]**, we can define a predicate-defined subset BlueCars as all members of Cars with a value of "blue" for their color property. A predicate-defined subset has the same type as its superset (i.e. BlueCars has type **Set[Car]**). We call a predicate-defined set, a ***class***. Classes give us the ability to separate constrained sets, requiring run-time membership tests, from the type system that is reserved for compile-time checking. It is important to note that while a subtype relationship between two types induces a subset relationship between their extents, a subset relationship between two classes does not necessarily imply a subtype relationship.

A class definition can be defined to either *constrain* its membership or provide a *view* of its parent class. If a class C defined by predicate P is defined as a *constrained-class*, then any attempt by a transaction T to change the state of c, an element of C, such that P(c) is false will cause T to abort. A class C defined by predicate P may also be defined to be a *view*. If the state of some object c in class C changes such that c no longer obeys predicate P, c is automatically removed from the class. Object c may now satisfy some other predicate P′ that defines class C′, in which case it is automatically inserted into C′.

An instance x of a type T has a unique identity that is independent of the state of x. Although an object's identity may be implemented by a system supplied object identifier, the existence of the identifier

is transparent to the language interface. Logically, the language always sees an object, and must access that object through the interface defined for its type. Object identity allows the implementation of references between objects. When an object x refers to an object y by means of a method M (i.e., M(x)=y), applying M to x produces object y (and does **not** give access to an identifier for y).

Properties in ENCORE reflect the abstract state of an object. The notion of *property* is modelled by one or more of the methods defined on a type. A property value is accessed by a special observer method which is required to have no side-effects on the observable state of the object. This method for property P is called Get_P. The Get_P method can return the value of a stored field or it may perform a more sophisticated computation based on the stored representation of the object. A property P may also support another function called Set_P that allows the value of P to be changed.

ENCORE, like many object-oriented programming languages [1], treats many parts of the language itself as objects. This includes types, methods, and properties. In particular, the treatment of properties as objects complicates the discussion in the previous paragraph. This reflective capability is useful to allow the database system to manage its own metadata. We will not discuss this capability any further here.

3 Querying in ENCORE

The ENCORE Query Algebra (EQUAL) is a collection of operators, defined for parameterized type **Set**, that can be used to construct queries over sets of ENCORE objects. A query is a request for information about the state of the database, thus our query operators provide an environment for gathering and organizing objects (and sometimes values, such as integers) in an ENCORE database. The operators support abstract data types and encapsulation by accessing objects only through the methods defined for their type, in particular the Get methods for properties. All queries are strongly typed, and can be statically type-checked.

The result of a query is a new database object. The creation of new objects leads to two questions: What is the type of the object? and What is its identity? Our requirement for static type-checking means that we cannot build new abstract data types. As a result, we define the type of any object built by a query to be **Set[T]**, where type **T** is either an existing database type or a parameterized **Set** or **Tuple** type. In this way EQUAL combines complex objects with abstract data types. The query operators build new set and tuple objects, intermixed in any order, above the abstract types. Unlike some complex object models, EQUAL treats sets and tuples as objects. Each object built by a query is a new object with a unique identity. This means that we can build alternative paths to the same object, and also that we might build objects that are considered to be duplicates. Our algebra does not build, however, recursive objects.

The support for object identity implies that we also need to support more than one notion of object equality. Two objects are *identical* when they are the same object (i.e. an object can only be identical to itself). We define a family of equality operations we call *i-equality* where i specifies the level, in a traversal of object structures, at which two objects refer to identical objects. We discuss these equalities in more detail, and also discuss some implications of support for object identity on query optimization, in [5].

As stated above, the EQUAL notion of a query is as an expression that, when evaluated, produces a new object. This is consistent with many programming language models that, for example, build a new array object every time a new declaration is encountered. Programs are free to mutate the object returned by a query at any time. Thus, we see that query results are not views. If we select, with a query, the blue cars from a set of cars, nothing prevents us from later placing a yellow car into this set. This query language characteristic implies that queries can be used as another kind of object creation mechanism in application programming.

EQUAL includes operations that are the analogs of relational algebraic operations as well as operations to manipulate the logical structure of ENCORE objects. The *Select*, *Project* and *Ojoin* (object join) operations are similar to their relational counterparts in meaning, but generalize the relational operations by applying functions (i.e. property methods or query operations) to the database objects. For S a Set and p a predicate, we define $Select(S, p) = \{s \mid (s\ in\ S) \wedge p(s)\}$. For S a Set, A_i a string, and f_i a function, $Project(S, [(A_1 : f_1), ..., (A_n : f_n)]) = \{[A_1 : f_1(s), ..., A_n : f_n(s)] \mid s\ in\ S\}$.

As an example, the query $Project(People, \lambda p\ [(P : p), (Toys : Select(p.Cars, \lambda c\ c.cost > p.salary)])$ builds a set of tuples each of which contains a **Person** object (P) and a **Set[Vehicle]** object (Toys).[2] For each p in the People set, a tuple is created with a P attribute whose value is p and a Toys attribute whose

[2] We assume here that People has type **Set[Person]** and that type **Person** has a Cars property that returns an object of type **Set[Vehicle]**. The λp is simply a way to generate a bound variable that will range over the members of People.

value is built by applying the Get_Cars method to object p. Note that in this example, the *Project* operation is similar to a relational *Outer-join*; each result tuple contains a **Person** object and may have an empty Toys attribute.

The application of a property to an object p (i.e. the application of a *Get_property* method) is a navigation from p to the object representing the value of the property. A navigation can be a single property retrieval, or a string of retrievals (e.g. p.mother.mother.cars to find one's grandmother's cars). Such navigation follows pre-defined relationships between objects. We also provide an *Ojoin* operation to explicitly join two sets of objects that are not necessarily related navigationally. For sets of tuple objects, *Ojoin* is analogous to a relational *Theta-join*. When a **Set[T]** is involved in an *Ojoin* (for **T** some abstract, non-tuple type), it is treated as a set of single-attribute tuples **Set[Tuple[A:T]]**. In other words, we define $Ojoin(S, R, A, B, p) = \{[A : s, B : r] \mid s \ in \ S \wedge r \ in \ R \wedge p(s, r)\}$ for S and R sets of objects of abstract types, A and B strings, and p a predicate. For example, an *Ojoin* of the set People with itself can be used to find pairs of people who are co-owners of a vehicle ($Ojoin(People, People, \text{"}Owner\text{"}, \text{"}CoOwner\text{"}, \lambda p_1, p_2 \ \ p_1.Cars \cap p_2.Cars$) results in a set whose members have type **Tuple[Owner:Person,CoOwner:Person]**). Our definition of *Ojoin* retains the associativity of the relational join, while still respecting the encapsulation of objects having abstract types.

Other operations that retrieve information are *Image*, *Union*, *Intersection* and *Difference*. *Image* is like a LISP mapcar; it allows the application of a function (which again may be a property or an EQUAL method) to each object in a set, collecting the results in a set object. We define $Image(S, f) = \{f(s) \mid s \ in \ S\}$ where S is a set and f is any function that can be applied to members of S. The *Union*, *Intersection* and *Difference* operations are like the relational operations, but also account for subtyping. An operation that combines a **Set[T]** with a **Set[Q]** results in **Set[R]** where **R** is the closest common supertype of **T** and **Q**. The equality test that naturally occurs in the definition of *Union*, *Intersection* and *Difference* can potentially add an extra level of complexity. If we assume that we always use the identical operation, then our definitions are the same as in most other models. But we could allow equality to be any of the other varieties of equality. This induces a family of *Union*, *Intersection* and *Difference* operations, one for each equality operator.

EQUAL also includes operations that manipulate the structure and identity of objects. The *Nest*, *UnNest* and *Flatten* operations work only with the structure of objects. *DupEliminate* and *Coalesce* manipulate identities. *DupEliminate* is necessary because we often build new tuple objects that may contain the same attribute values. The *Coalesce* operator is useful when we build new objects in a subquery. If two queries are executed, the results will be two distinct objects even though those objects may represent the same values. For example, suppose in the *Project* query matching people and their toys that two people are co-owners of the same two (expensive) cars. The query will build a new set object for each person, both sets containing the same two cars. A *Coalesce* operation could be used to ensure that both Toys attributes reference exactly the same set of cars.

Accessing and creating complex structures leads to the regular use of nested query expressions. In such expressions, query variables are not always used locally and can appear in nested scopes. For example, in the query matching people and their Toys, the variable p, representing a **Person** object, is nested in the predicate of the nested *Select* operation. The algebra also allows us to build flatter query expressions (using, for example, the *Ojoin* operator). The different types of expressions give us different opportunities for query optimization, so it is important that we be able to translate between them. Such transformations, however, are often context-sensitive since they involve arbitrarily nested scoping of query variables. As a result, the optimization of such expressions may be very difficult.

EQUAL generalizes relational operations by giving us the ability to access and produce encapsulated, logically complex objects. The algebra differs from most other proposed algebras for object-oriented systems in its treatment of abstract data types, encapsulated objects, and object identity. For more detailed information about the algebra see [6].

4 Research Directions

Although the basic ENCORE database system has been implemented, we are still extending it in a number of fundamental ways. These extensions are still under development at the time of this writing.

The optimization of queries is of major importance in a database system and we have found that an object-oriented database requires new optimization strategies to support features such as abstract data types, methods, and object identity. We are exploring a variety of strategies for query optimization to meet the

requirements imposed by our model and operators. For example, we are looking at strategies for transforming nested queries (such as the query about people and their toys in the previous section) and strategies that consider the effects of method cost on query transformation. Existing optimizers have a fixed set of strategies for controlling the optimization process. We have found that object-oriented database systems generate a need for many different optimization strategies. To manage this need, we are designing a new architecture for a query optimizer that can incorporate many different strategies for optimization and can be extended with new strategies. This architecture, although motivated by the object-oriented model, represents a novel approach to the optimization process for any database model.

The ENCORE model, like many type systems, supports constraints in a limited way by means of static type-checking and constrained collections. These facilities do not provide the kind of global constraints that one would like to express in a database. Global constraints span multiple types. For example, the classic constraint, *No Employee can make more than his/her manager*, involves the types **Employee**, **Manager**, and **Department**. The **Department** type is involved indirectly since an employee *e*'s manager can only be discovered by navigating from *e* to *e*'s department *d*, and from *d* to *d*'s manager. We are looking at techniques for taking constraints, such as the one expressed above, and decomposing them into a set of conjuncts such that each conjunct involves a single type T and together the conjuncts are equivalent to the original constraint. In this way, the conjuncts can only be violated by invoking an method of T on one of its instances. This approach limits the amount of checking that is required on database update.

View definition is an area in which most object-oriented database systems suffer. We have been studying the requirements for a view definition facility that could be added to the ENCORE model. Our current approach allows a *view* to be defined over a database, called the *base*, by means of a combination of data abstraction and queries [2]. The query returns a set of objects from the base that must conform to the type of the representation of an abstract type that is available in the view. In other words, the query provides the representation for the view type and the data abstraction allows the view definer to layer arbitrary new interfaces over this representation. We are currently looking at efficient ways to implement this idea.

An ENCORE prototype has been implemented on Sun 4 and SparcStation platforms running SunOS 4.1.1. It makes use of the ObServer object storage system, also developed at Brown.

References

[1] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[2] Sandra Heiler and Stanley Zdonik. Object Views: Extending the Vision. In *Proceedings of the 6th International Conference on Data Engineering*, pages 86–93, 1990.

[3] Paris Kanellakis, Christophe Lécluse, and Philippe Richard. Introduction to the Data Model. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, CA, 1991.

[4] Barbara Liskov et al. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[5] Gail M. Shaw and Stanley B. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 264–278, December 1989.

[6] Gail M. Shaw and Stanley B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the 6th International Conference on Data Engineering*, pages 154–162. IEEE, 1990. An early version of this paper appears as Brown University tech report CS-89-19.

[7] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the Hawaii International Conference on System Sciences*, January 1986.

Query Optimization in Revelation, an Overview*

Scott Daniels,[1] Goetz Graefe,[2] Thomas Keller,[1]
David Maier,[1] Duri Schmidt,[1] and Bennet Vance[1]

Introduction

The Revelation project is an experiment in query optimization in object-oriented databases [Graefe88]. Our goal is to expand query optimization and processing technology to address and exploit new modeling and query extensions. Since encapsulation (an important feature in object-oriented systems) works against query optimization, we provide a system component that may cross the encapsulation boundary and reveal implementation data. Our project draws its name from that component, the *Revealer*.

The single feature early users of next-generation database prototypes and products value most is type extensibility—the ability to add their own application-specific data types to the DBMS, giving more natural building blocks for modeling. It also provides logical data independence, allowing data reorganization or reimplementation of behavior while providing a constant interface to applications. Further, user-defined data types are well suited to re-usability, combining data structure and operations in logical units.

Some commercial products already support some user extension of system types, but we believe users want more potent type extensibility, as suggested in "The Object-Oriented Database Manifesto" [Atkinson90]. User-defined types should be *first-class, immediate,* and *abstract*. Instances of *first-class* types can appear wherever instances of system-supplied types can. *Immediate* type definition is available as readily as schema modification and does not require rebuilding the DBMS. This implies that types should be defined in the Data Definition and Manipulation Languages, rather than the implementation language of the DBMS. An *abstract* type can hide its implementation; data and function can be encapsulated from clients of the type. It is desirable (and possibly vital to query optimization) that the functions associated with a type be understood by the database system, and not be simply black-box routines invoked at appropriate moments.

We have accordingly concentrated Revelation's efforts on handling modeling features that support user-defined data types, retaining as much set-processing technology from relational databases as we can. Features that provide this support include encapsulation, complex state, object identity, polymorphism, ordered data structures (such as vectors, matrices, and grids), and specification and implementation hierarchies.

Modeling Features and Their Impact on Query Optimization

Imagine a database for scene description, as might be used for rendering images or motion planning. One type in this database is `Polygon`, a basic building block for scenes. The behavior of a polygon includes returning an ordered list of its vertices, computing its intersection with a given line or plane, and moving its vertices. It has subtype `ColoredPoly` which has its own subtype, `PatternedPoly`.

Encapsulation allows multiple implementations of a type to co-exist in the database. Consider various possible implementations of `Polygon`—an explicit list of `Point` values, a list of references to `Point` objects, or a reference to another `Polygon` and a transformation (such as scaling or rotation) to apply to all its `Points`. One can imagine using several implementations simultaneously (for different `Polygons`) within the same database. Encapsulation says the particular choice can be hidden from clients of the type, because each supports the same operations. It also gives control over certain invariants during update. If a `Polygon` is represented as a list of points that applications can modify directly, it is difficult to enforce invariants such as coplanarity and convexity. If the update operations for a `Polygon` are encapsulated with its representation, then invariants can be checked there, rather than in the application code. Current query optimization assumes that a complete description of the query is available, in terms of structural operators. Encapsulation, with its separation of behavior from structure, implies that the structural component may not be discernible during query optimization. It also causes problems with defining and maintaining indices, as well as making

* This research was supported in part by National Science Foundation grants IRI-89-20642 and IRI-89-12618.

1 Computer Sciences and Engineering, Oregon Graduate Institute, 19600 SW Von Neumann Dr., Beaverton, OR 97006-1999 (daniels, tkeller, maier, schmidt, bennet)@cse.ogi.edu

2 Department of Computer Science, University of Colorado, Boulder, CO 80309-0430 graefe@cs.colorado.edu

it difficult for the optimizer to get the "big picture;"- with encapsulation, a single message expression may hide an arbitrary amount of data manipulation. With just a small expression to optimize, the range of transformations to apply is limited. Yet another problem with encapsulation comes with multiple implementations of a type. Current set processing depends a great deal on homogeneity of structure, for example, in allocating temporary space for records in intermediate results, or in computing offsets for record fields.

Complex State means a rich set of structuring mechanisms is available for specifying the representation of a type, and that these constructors can be freely composed. Thus, the representation of a `Polygon` in one implementation might be `(offset: (X, Y, Z: Integer), vertices: list of (X, Y, Z: Integer))`. The list `vertices` captures the points that define the shape and the triple `offset` positions the polygon in 3-space. Note the nesting of the tuple constructor inside the list constructor inside the tuple constructor. However, only a small part of that state may be needed for a particular query. Reading the entire state of an object (especially if "entire" includes referenced objects) can fill memory with irrelevant information. Reading pieces of an object only as demanded for computation may not be efficient, either.

Object Identity allows shared references to subcomponents of an object irrespective of their values. We might want to have an implementation of `Polygon` that uses `list of Point` as a representation, where `Points` are objects. Two polygons can share points in their representations, in order, say, to keep a edge in common while updates are made to underlying points. The problems this causes for query optimization resemble those of complex state; straightforward implementations quickly degenerate into pointer chasing on disk.

Hierarchies exploit similarities among classes of entities, for type specification or implementation. A specification example assumes `ColoredPoly` (a subtype of `Polygon`) responds to a `color` message, as does `PatternedPoly` (a subtype of `ColoredPoly` and therefore `Polygon`). If a `Polygon` responds to a message asking for its surface area, then `ColoredPoly` and `PatternedPoly` must also respond to that message. Thus, a heterogeneous set of `Polygon`, `ColoredPoly`, and `PatternedPoly` can be uniformly queried to select those larger than a certain area, since that is behavior common to all three types. Similarly, a set of `ColoredPoly` and `PatternedPoly` could be uniformly queried to select red ones, though a simple `Polygon` could not be included in such a set and maintain correct typing. Type hierarchies allow heterogeneous collections of objects to be queried on their common protocol. This means the language bears a *polymorphic* interpretation that reduces the efficiency of set processing. This polymorphism (much like encapsulation) causes problems in defining and maintaining auxiliary access structures.

Many of the new data models provide more general *persistent name spaces* than conventional data models. These name spaces resemble those of programming languages, with variables declared of arbitrary types, and arbitrary numbers of variables of a given type. Moreover, the variables can have their values reassigned. Contrast this situation with that in relational databases, where the persistent variables may only be relations (as opposed to tuples or scalars). One major problem for optimization is where to attach statistics that an optimizer would use in cost estimation. In a relational system, for example, relation names are statically bound to relation instances, and there is no distinction between associating statistics with a relation or its name. With a set-valued variable that can be reassigned during a transaction, it seems that statistics should be associated with set instances. However, it is the variable name that is available during query optimization.

In examining ordered structures, we see the "iteration idioms" supported by operators in record processing systems (such as select and join) are not expressive for common manipulations on scientific data types. Biochemists are interested in correlating patterns of amino acids with structural features. Supporting such pattern searching efficiently at the physical level means scanning the underlying list structure with a window of elements. Typical physical scans in relational implementations are built for record-at-a-time access.

The Revelation Approach

In the Revelation project we are looking at approaches to deal with each of the problems above.

For encapsulation, we introduce a *Revealer,* a trusted system component that is allowed to break encapsulation in order to expand messages into more detailed expressions. It provides the Optimizer with more transformation choices and thus a larger scope for planning the query. To deal with hierarchies and polymorphism we employ two approaches. One approach looks across multiple implementations for "coincidences"— commonalities of definition. The other is an algebra operator that partitions a collection into multiple data streams based on type. For ordered structures, we want our object algebra to contain operators that can

represent their most common iterators. We are looking at scientific codes and statistical analysis packages such as S for insights here. To avoid the inefficiencies of object-at-a-time access from naive handling of complex objects and object identifiers, we add the "assembly" operator for query evaluation.

Finally, in dealing with a more general naming scheme for persistent data items, we are tracking stability of bindings in our name space. Thus, a variable name can be bound more or less statically to a type, a constraint, an implementation, an object identity, an auxiliary access path, a statistic, or even a state. The query optimizer can incorporate this information based on the expected lifetime of a query (e.g., one transaction, one execution of an application program, or multiple executions of an application program).

Query Architecture

The Revelation architecture has four levels. The top level consists of the *Interpreter* and *Schema Manager.* The schema language is used to define type interfaces (protocols) and their implementations [Daniels90]. It also provides a name space in which to declare persistent variables. The Interpreter can naively evaluate expressions, or alternatively it can pass an expression to the next level, the *Revealer.* The Revealer expands an expression into a tree in an object algebra, through replacement of operations by their methods, and obtaining information about bindings and statistics from the name space. The resulting algebra tree, with possible annotations, is passed to the third level, the *Optimizer.* The Revelation query optimizer will be produced by a second-generation optimizer generator based on the EXODUS Optimizer Generator [Graefe87]. The generator takes rules involving algebraic equivalences and cost metrics and produces an optimizer incorporating that knowledge. The Optimizer produces a query plan, which is a program for the Revelation *Query Evaluator,* the fourth level.

Interpreter and Schema Manager

A Revelation schema has three parts: protocols, implementation and the name space. Protocols describe the interfaces to database elements in terms of the permitted operations on instances and give signatures to those operations—protocols (types) for arguments and the result. Note that nothing about data structures or layout is included in a protocol. Protocols are related in a hierarchy by conformance, where if protocol `ColoredPoly` conforms to protocol `Polygon`, an instance of `ColoredPoly` can be substituted where an instance of `Polygon` is expected. Implementations consist of a representation and method definitions. Representations for instances are constructed by free composition of data structures, such as `integer` and `array(T)`. Methods are defined in a language that extends the object algebra with control structures and assignment. We allow one implementation to satisfy several protocols, and one protocol may be satisfied by multiple implementations. The name space contains persistent variables that may be typed by any of the defined protocols. These variables may have properties besides types statically bound to them, such as implementation or even state. The name space is where constraints such as referential dependencies and subset relationships are defined.

The Schema Manager keeps track of protocol and implementation definitions and declarations in the name space, supplying information as requested by the Interpreter and Revealer. It can also track information relating to the relative permanence of the bindings and statistics on the numbers of instantiations of each implementation in the database. The Interpreter can execute expressions (queries) involving the persistent variables in a straightforward manner. Each object carries a reference to its implementation. When the Interpreter encounters an operation on an object, it looks up the method for that operation in the appropriate implementation and evaluates the expressions in it recursively.

Revealer

If the Interpreter passes an expression to the Revealer, the Revealer attempts to expand that expression by incorporating information from the schema and name space. The reason for the expansion is to gain more foreknowledge of the computation steps and data elements used in a query. The expression as given may contain operations whose methods are encapsulated from its issuer, and leaving them encapsulated limits the choices for transformations available to the query optimizer. The Revealer may break encapsulation on operations, expanding them to their methods where possible. The goal of the Revealer is to expand as many message nodes in the object algebra tree as possible. Complete expansion may not be possible because of differing implementations per protocol or because general control constructs such as recursion prevent

it. In such cases, expressions are passed to the Optimizer as unexpanded subtrees, to be evaluated by the Interpreter at run time. In some cases, even though the method for a message cannot be determined, the Revealer may be able to provide partial information about such a subtree, such as the portion of the object's representation accessed by the method or whether the method is read only.

The conversation between the Revealer and the Schema Manager is through the *Annotater*. A collection of annotation kinds are defined for nodes in an expression tree, such as protocol, implementation, representation, and accessed structure. The Revealer makes requests of the Annotater to derive a particular annotation for a specific expression node, which deduces and records the value of the annotation. The Annotater may return a collection of values, due to ambiguity arising from multiple implementations of a single protocol. Expansion of an operation to its implementing method is one such annotation. The Revealer can specify the level of stability the Annotater should assume when deriving annotations, such as whether to assume the set of implementations or protocols is fixed. Here is where the notion of *coincidences* comes in. If, for example, we are optimizing assuming no change in implementations, the Annotater can determine all possible implementations of a particular protocol. If there is only one implementation (or if there are several that all share a single method), it may expand a message into the corresponding method body. This may convert the access to a simple structural access. We note that the Annotater operates on a single expression node at a time—the Revealer controls the order and extent of expansion.

Optimizer

The expanded expression tree is handed to the Optimizer for optimization and transformation to a query plan. The expression tree is mostly operators from the object algebra, with possibly embedded Interpreter invocations on schema-level types. The object algebra is structural, operating on values formed by free composition of a fixed set of constructors, which will contain at the least base types (such as integer and string), tuples, multi-dimensional arrays, bags, and object references. In addition to the operators of relational algebra such as select and join, we include operators that capture common control patterns of scientific data manipulation like matrix algebra operations, time-series averaging, and time-step computations on grids. The Optimizer itself will be produced by an optimizer generator and providing rules for algebraic identities on algebra operators should be straightforward. However, rules for translating algebra trees into query plans and cost estimates will require more work.

A *query plan* is a directive as to what physical operators to run, where to run them and in what order to run them. The physical operators correspond largely to those in the logical object algebra in terms of the functions they compute. However, the physical operators use different algorithms to realize those functions, and their execution costs depend greatly on dataset sizes and layouts. We are undertaking a methodical examination of physical properties of data and their interaction with different physical operators. Examples of physical properties are clustering, partitioning, size, auxiliary access paths, and location in the memory hierarchy. We categorize physical operators by whether they *require, preserve, enforce,* or *destroy* a particular property. For example, a nested-loops join might preserve a certain sort order, while hash join will destroy it. We are looking at how to construct cost functions that incorporate information on these data properties.

Query Algebra

Our query algebra must include bulk operations, beyond the traditional set operations, for ordered data such as lists and matrices. We also need algebraic identities on these new operators, for query optimization. Matrix algebra is an obvious starting point, but we anticipate other identities.

In examining scientific applications, we have seen other common classes of operations on ordered structures that we should support. One is conversions between bulk types. An example involves a relation with attributes `(Altitude, Temperature, Pressure)`. We want to sort on `Altitude`, and then project `Temperature` and `Pressure` into two parallel one-dimensional arrays to provide input for statistical operations. Relational query languages don't work on this query since, although most can express sort orderings, they don't guarantee order preservation after the sort key is projected away. Another common manipulation is structural selection such as taking a *slab* or *pencil* from a multi-dimensional array. We are seeking a small set of operations on ordered structures that expresses a large variety of the desired manipulations.

Another query algebra problem is subtyping, for which we hope to use discriminated unions [Vance91]. A powerful feature of object-oriented systems is the late binding of message names to methods. An object-oriented database will complete this binding as much as possible before evaluating a query, but the ambiguity

of message names at optimization time still presents a challenge. Because of substitutability, a set over type **Polygon** may include objects both of type **Polygon** and type **ColoredPoly**. Suppose a query over such a set sends a message **area** to each object in the set. If the same method for **area** is used by all **Polygon** and **ColoredPoly** objects, references to **area** may safely be replaced with that method body. This extends the query expression tree and permits optimizations across the method boundary.

If **ColoredPoly** provides a different method for **area**, this optimization is invalid. However, if the **Polygon** and **ColoredPoly** objects can be distinguished, the operations on the two cases can be optimized separately. Internally each object has a tag indicating its implementation. A query evaluator, cooperating with a query optimizer, can use this tag to execute different optimized query plans for the different kinds of objects.

In the algebra we use discriminated unions to distinguish objects of different but conforming types. Thus, a set over type **Polygon** is represented in the algebra as a set over type (**ColoredPoly** + **Polygon**), and each object in the set is injected into this type. Using this abstraction does not change the way a set over type **Polygon** is stored, but it offers conceptual advantages. At times it may be beneficial to split a set of (**ColoredPoly** + **Polygon**) into sets of **ColoredPoly** and **Polygon**, and then operate on those sets separately. The use of the union type allows this *split* operation to be expressed algebraically. Moreover, the result sets contain strictly monotype, untagged objects, affording more efficient processing.

Query Evaluator

The Query Evaluator executes query plans from the Optimizer. It is based on the Volcano extensible query execution software [Graefe90], which implements the physical operators and handles data flow between them. We have used Volcano's extensibility to add a pair of operations for these new access forms. The *assembly* [Keller91] operator supports complex state. It takes a template that describes some fragment of an object's state and a set of object references and then assembles the required pieces of each referenced object in main memory. Unlike the naively ordered access of the Interpreter, the assembly operator re-orders the accesses to take advantage of clustering. Another operator we have used to extend Volcano invokes the Interpreter during query execution, thus resolving unrevealed subtrees passed through the Optimizer.

Conclusion

We have described a number of features of next-generation databases, and shown both their utility and the challenges they present to query optimization. We have also laid out our basic attack on these challenges. We expect to complete our first prototype system this summer, and use it as a testbed for our optimization ideas. While the system we are building has an object-oriented model, we expect this work to provide useful insights to anyone trying to improve query optimization in databases with user-extensible types.

[Atkinson90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, "The Object-Oriented Database Manifesto," Deductive and Object-Oriented Databases, W. Kim, J.-M. Nicolas, and S. Nishio editors, Elsevier Science Publishers B. V. (North-Holland), 1990.

[Daniels90] S. Daniels, "Speaking in Tongues: The Language of Revelation," OGI TR CSE-91-007, April 1990.

[Graefe87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the 1987 SIGMOD Conference*, San Francisco, CA, May 1987.

[Graefe88] G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: a Prospectus," *Advances in Object-Oriented Database Systems,* Lecture Notes in Computer Science 334, Springer-Verlag, September 1988.

[Graefe90] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Proceedings of the 1990 SIGMOD Conference,* Atlantic City, NJ, May 1990.

[Keller91] T. Keller, G. Graefe, and D. Maier, "Efficient Assembly of Complex Objects," to appear in *SIGMOD 91,* also OGI TR CSE-90-023, April 1991.

[Vance91] B. Vance, "Towards an Object-Oriented Query Algebra," OGI TR CSE-91-008, May 1991.

IEEE Computer Society
1730 Massachusetts Avenue, NW
Washington, DC 20036-1903

Mr. Henry F. Korth
University of Texas
Taylor 2124 Dept of CS
Austin, TX 78712
USA