[23] U. Nanni, S. Salza, and M. Terranova. An Algebraic Approach to the Manipulation of Complex Objects. *Proceeding Hawaii International Conference on System Sciences*, Kaloa, Hawaii, January 7-10, 1992.

[24] U. Nanni, S. Salza, and M. Terranova. The LOGIDATA+ Prototype System. *Technical Report of the International Computer Science Institute*, TR-92-007, 1992.

[25] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Trans. on Database Syst.*, 13 (4), 389–417, December 1988.

[26] H.-J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 1986.

[27] M.H. Scholl and H.-J. Schek. A relational object model. *Proceedings International Conference on Data Base Theory*, Paris, *Lecture Notes in Computer Science*, 470, 89–105, 1990.

[28] J. D. Ullman. *Principles of Database and Knowledge-Base Systems (vol. I).* Computer Science Press, 1988.

[8] F. Bancilhon and S. Khoshafian. A calculus for complex objects. *Journal of Comp. and System Sc.*, 38(3):326–340, 1989.

[9] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. *Proceedings ACM SIGMOD International Conf. on Management of Data*, 225–236, 1990.

[10] S. Ceri, S. Crespi-Reghizzi, G. Lamperti, L.A. Lavazza, and R. Zicari. ALGRES: an advanced database system for complex applications. *IEEE Software*, 7(4), July 1990.

[11] S. Ceri, G. Gottlob and L. Tanca. *Logic Programming and Databases.* Springer Verlag, 1990.

[12] A. K. Chandra. Theory of Database Queries. *Proceedings ACM Symposium on Principles of Database Systems*, 1–9, 1988.

[13] L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. *Proceedings ACM SIGMOD International Conf. on Management of Data*, 273–283, 1989.

[14] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13 (6), 377–387, 1970.

[15] P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. *Proceedings IEEE Computer Software Applications*, pages 464–475, 1983.

[16] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32, North Holland, 1986, 265–280. Also in *Proceedings Int. Conf. on Foundations of Computer Science*, 346–353, 1985.

[17] R. Hull and J. Su. On accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions. *Proceedings ACM SIGACT SIGMOD Symposium on Principles of Database Systems*, 147–158, 1989.

[18] G. Jaeschke and H.-J. Schek. Remarks on the algebra for non first normal form relations. *Proceedings ACM SIGACT SIGMOD Symposium on Principles of Database Systems*, 124–138, 1982.

[19] S. Khoshafian and G. Copeland. Object identity. *Proceedings ACM Symposium on Object Oriented Programming Systems, Languages and Applications*, 1986.

[20] G.M. Kuper. *The Logical Data Model: A New Approach to Database Logic.* PhD thesis, Stanford University, 1985.

[21] G.M. Kuper and M.Y. Vardi. A New Approach to Database Logic. *Proceedings Third ACM SIGACT SIGMOD Symposium on Principles of Database Systems*, 1984.

[22] G.M. Kuper and M.Y. Vardi. On the complexity of queries in the logical data model. *Proceedings International Conference on Data Base Theory*, 267–280, 1988.

7 Conclusions

In this paper we present LOA, that is an algebra for the manipulation of complex objects. The algebra refers to a data model (LOGIDATA+) dealing with structured data types and two different kinds of data collections: classes, with object identity enforcement, and value based relations. Moreover, due to the inclusion of a FIXPOINT operator, using LOA it is possible to express recursive programs.

With respect to the basic relational primitives, more powerful conditions are provided, based on a set of navigational operators that allow to move through complex data structures; these are to be used in conjunction with the classical operators of projection, selection and join. Additional new operators are defined to manipulate the structure of data, and to attain the full capability of data restructuring.

The algebra has been developed within the project LOGIDATA+ as an internal language in the prototype object oriented system. The algebraic set-oriented approach proves to be effective, especially when moving to the object oriented model traditional database applications, where transactions typically operate on large sets of objects.

In the prototype system, user programs are to be compiled into LOA programs which are interpreted by an underlying object manager. Details of the implementation of the object manager are described in [24].

Although the algebra has been originally conceived as an internal language for the prototype, it proves also to be quite effective in expressing complex queries. It may then form the basis for the definition of a high level user oriented query and manipulation language.

References

[1] S. Abiteboul and C. Beeri. *On the Power of Languages for the Manipulation of Complex Objects.* Technical Report 846, INRIA, 1988.

[2] S. Abiteboul and N. Bidoit. Nonfirst normal form relations: an algebra allowing data restructuring. *Journal of Comp. and System Sc.*, 33 (1), 361–393, 1986.

[3] S. Abiteboul and S. Grumbach. Col: a logic-based language for complex objects. *Proceedings International Conference on Extending Database Technology)*, Venezia, *Lecture Notes in Computer Science*, 303, 271–293, Springer-Verlag, 1988.

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. *Proceedings ACM SIGMOD International Conf. on Management of Data*, 159–173, 1989.

[5] R. Agrawal. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. *IEEE Transactions on Software Engineering*, 14 (7), July 1988.

[6] A. V. Aho and J. D. Ullman. Universality of Data Retrieval Languages. *Proceedings ACM Conference on Programming Languages*, 110–120, 1979.

[7] P. Atzeni and L. Tanca. The LOGIDATA+ Rule Language. *Proceedings Workshop "Information Systems '90"*, Kiev, 1990.

More specifically NEST is used to collect in a set the group of distinct values of an attribute of a tuple that share the same value of the rest of the tuple. Formally, if $\mathcal{R}$ is a relation of type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$ then $\mathcal{R}' \Leftarrow \text{NEST}(\mathcal{R}; \mathcal{A}_k)$ is a relation of type $\tau' = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \{\theta_k\})$.

UNNEST is the reverse operation, and generates a separate tuple in the result relation for every distinct value in an attribute of type set. Formally, if $\mathcal{R}$ is a relation of type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \{\theta_k\})$ then $\mathcal{R}' \Leftarrow \text{UNNEST}(\mathcal{R}; \mathcal{A}_k)$ is a relation of type $\tau' = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$. With the same definition the UNNEST can be applied also to sequences.

For example the following steps compute a binary relation that associates to each father the set of his wives:

```
Couples ⇐ PROJECT(Family; (father,mother))

Polygamy ⇐ NEST(Couples; mother)
```

On the other end the following steps compute from `Family` the relation `Parent` that contains the couples (parent, child):

```
U_Family ⇐ UNNEST(Family; children)

Parent ⇐ UNION(PROJECT(U_Family;(father:Person,children:Person)),
               PROJECT(U_Family;(mother:Person,children:Person)))
```

In addition to NEST and UNNEST two more operators are provided, CLUSTER that groups a subset of attributes in a tuple to generate a subtuple in a single attribute, and MELT which does the reverse operation, i.e. flattens a tuple structure with a nested tuple attribute.

Conversion operators basically allow to convert values into objects and vice versa. Depending if the whole structure or part of it is to be converted, different operators are required:

- CLASS: transforms the elements of a relation into objects, thus defining a new class and the corresponding object type.
- DECLASS: transforms the objects in a class in structured values, by depriving them of the object identity, and thus eliminating duplicate values from their result.
- OBJECT: converts a value component inside a tuple structure into a component of object type, thus defining a new class.
- VALUE: converts an object component inside a tuple structure into the corresponding value.

Note that the execution of the conversion operators may require the invention of object identifiers.

- If $\mathcal{R}$ is a relation of type τ and $\tau \preceq \tau'$ then $\mathcal{R}' \Leftarrow \text{PROJECT}(\mathcal{R}; \tau')$ is a relation of type τ'.
- If $\mathcal{C}$ is a class of type $\omega = [\mathcal{C}, \tau]$ and $\tau \preceq \tau'$, then $\mathcal{C}' \Leftarrow \text{PROJECT}(\mathcal{C}; \tau')$ is a class of type $\omega' = [\mathcal{C}', \tau']$.

The projection includes the duplicate elimination. This can be effective only when the operand is a relation.

The JOIN is defined in such a way that, regardless of the type of the operands, that can be classes or relations in any combination, the result is a binary relation, i.e. one further level is added to the structure of the type.

Formally, if $\mathcal{X}_1$ and $\mathcal{X}_2$ are classes or relations of types θ_1 and θ_2, and f is a condition involving components from types θ_1 and θ_2, then $\mathcal{R}' \Leftarrow \text{JOIN}(\mathcal{X}_1, \mathcal{X}_2; f)$ is a relation of type $\tau' = (\mathcal{A}_1 : \theta_1, \mathcal{A}_2 : \theta_2)$.

With reference to the schema in Fig. 1 the following are sample queries:

```
Homonyms ⇐ SELECT(Family;Family.father.name.SET(givennames) ∩
           ∩ Family.FLAT(SET(children.name.SET(givennames)))≠ ∅
```

`Homonyms` contains all the families in which at least one child shares a name with the father.

```
F_ names ⇐ PROJECT(Family; (father:(name:(givennames)))).
```

`F_names` contains the givennames of the fathers in `Family`. Note that, as the type of the result is a supertype of the type of the operand, it maintains the complete nested structure around the projected attributes. To flatten the structure, one should use the structure operators introduced in the next section.

```
Artists ⇐ JOIN(Family,Student;(Family.SET(children) ∋ Student) ∧
          ∧ (Student.curriculum.exams.course ∋ 'fine arts')).
```

`Artists` is a binary relation of type: (`Family` : $\tau_{\texttt{fam}}$, `Student` : $\omega_{\texttt{stud}}$) and contains all the families with a child who passed the exam of fine arts.

Additional operators are provided for set operations. These are straightforward extensions of the corresponding relational operators UNION, INTERSECT, DIFFERENCE.

6 Structure and Conversion Operators

The need for these operators arises from the richer structure of the types in the object oriented data model. They are required in the algebra to attain the full capability of data restructuring. This includes the transformation of the components in a structure, from values to objects, and vice versa. Some of these operators have been introduced in algebras for the nested relation model [25, 10].

The structure operators apply only to relations, and produce a relation as a result. They allow to modify the structure by means of actions such as gathering several attributes in a tuple and multiple values of an attribute in a set.

- `Student`: denotes an object of type $\omega_{\texttt{stud}}$ in the class `Student`.
- `Student.curriculum.exams`: denotes the set of couples (`course`, `degree`) of an object of the class `Student`.
- `Family.children.name.givennames`: denotes the sequence of sequences of givennames of the children of a given family. To get the set of names one should write:
  `Family`.FLAT(SET(`children.name`.
  SET(`givennames`)))

Components are used to build *conditions*, i.e. boolean predicates based on the comparisons between components and constants of a compatible type. Formally:

- If E_1 and E_2 are components of value type τ_1 and τ_2, with value sets $\mathbf{V}_1 \equiv \mathbf{V}_2 \equiv \mathbf{V}$, then $E_1 \text{OP} E_2$ and $E_1 \text{OP} v$ are conditions, where OP is a comparison operator, and $v \in \mathbf{V}$.
- If E_1 and E_2 are components of object type ω_1 and ω_2, then $E_1 = E_2$ or $E_1 \neq E_2$ are conditions.
- If E_1 is a component of type θ and E_2 is a component of type $\{\theta\}$ then $E_1 \in E_2$ and $E_1 \notin E_2$ are conditions.
- Every boolean expression whose terms are conditions is a condition.

Note that only equality and inequality comparisons are allowed between components of object type, and that comparison may take place between objects of different types. In all these cases the comparison is based on the object identity.

5 Extending the Operators of Relational Algebra

The traditional operators of the relational algebra, SELECT, PROJECT and JOIN, are extended in two ways. First more powerful conditions are allowed, using the framework introduced in the previous section. The other extension concerns the definition of the type of the result according to the type of the operands and the structure of the conditions.

The SELECT extracts from a relation or a class the subset of elements satisfying a given condition. Formally:

- If $\mathcal{R}$ is a relation of type τ and f a condition on the type τ, then $\mathcal{R}' \Leftarrow \text{SELECT}(\mathcal{R}; f)$ is a relation of type τ.
- If $\mathcal{C}$ is a class of type $\omega = [\mathcal{C}, \tau]$ and f a condition on the objects of $\mathcal{C}$, then $\mathcal{C}' \Leftarrow \text{SELECT}(\mathcal{C}; f)$ is a class of type $\omega' = [\mathcal{C}', \tau]$.

The PROJECT extends the relational operation in the sense that, instead of a subset of attributes one has to specify the supertype of the operand type that contains the required information. Note that all the nested levels of the data structure are kept in the result. Formally:

In the relational algebra, *selection* and *θ-join* operators require as additional argument a *clause* or *condition*, that is a predicate which must be satisfied by the tuples to contribute to build the resulting relational table. In the most general case a predicate is a boolean formula over simple predicates, and each of them has an operator (a binary predicate), and two atomic values as operands, which are either two attributes of the tuple(s) or one attribute and a constant (only in the case of a selection clause).

In the more complex LOGIDATA model, a clause is used in selection and join as well, and it has the same basic structure, consisting of a boolean formula over simple predicates. In this case the operands of the simple predicates are not necessarily atomic values but can have a structure of arbitrary complexity. We introduce the notion of *derived component*, which is an expression that can be used in the conditions to denote an operand extracted from an object or a tuple: in general it is a structured value that is a part of it or can be built up from it.

Formally, given a relation $\mathcal{R}$ or a class $\mathcal{C}$ of type θ:

- If $\mathcal{A}$ is an attribute of type θ then $\mathcal{A}$ denotes a derived component from $\mathcal{A}$ of type θ.
- If $\mathcal{A}$ is an attribute of type θ, with corresponding value type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{A}.E_i$ is a derived component from attribute $\mathcal{A}$ of type θ_E.
- If $\mathcal{A}$ is an attribute of type $\{\theta_B\}$, with corresponding value type $\tau_B = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{A}.E_i$ is a derived component from attribute $\mathcal{A}$ of type $\{\theta_E\}$.
- If $\mathcal{A}$ is an attribute of type $\langle\theta_B\rangle$, with corresponding value type $\tau_B = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{A}.E_i$ is a derived component from attribute $\mathcal{A}$ of type $\langle\theta_E\rangle$.
- If E denotes a derived component of type θ_E, with an associated value type τ_E, then:
  - if $\tau_E = \{\{\theta_B\}\}$ then FLAT(E) denotes a derived component of type $\{\theta_B\}$;
  - if $\tau_E = \langle\langle\theta_B\rangle\rangle$ then FLAT(E) denotes a derived component of type $\langle\theta_B\rangle$;
  - if $\tau_E = \langle\theta_B\rangle$, then POS$(E, n)$ denotes a derived component of type θ_B (the n-th element in the sequence), and SET(E) denotes a component of type $\{\theta_B\}$;
- If $\mathcal{R}$ is a relation of type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{R}.E_i$ is a derived component from the relation $\mathcal{R}$ of type θ_E, and $\mathcal{R}$ denotes a derived component of type τ, i.e. a tuple in the relation.
- If $\mathcal{C}$ is a class of type θ with associated value type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{C}.E_i$ is a derived component from the class $\mathcal{C}$ of type θ_E, and $\mathcal{C}$ denotes a derived component of type θ, i.e. an object in the class.

With reference to the schema in the Figures 2 and 3, the following are examples of components:

We remark that the presence of several fixpoint blocks (and several equations within each block) should not increase the expressive power of the language (see [16] for the case of a flat model) but makes more flexible and natural the description of a LOA program and furthermore simplify the compilation process, that is the translation of user queries from the external language to LOA.

In the seminal paper [6] several ways of introducing a fixpoint operator (or equivalent constructs) within a database language are considered. This has been done with quite different approaches such as, for example the *alpha* operator [5] or the *pointwise recursion* [17]. In [12] an overview of the topic is given, comparing the expressive power and complexity of the various constructs. Examples of an approach integrating a rule based language and an object oriented data model are IQL [4], and LOGRES [9]. An algebraic approach integrating recursion and a complex data model can be found in [13].

In the following sections we specify the syntax and semantics of the operators used in LOA expressions.

As far as the algebraic operators are concerned, we generalize previous proposals. These basically refer to the Nested Relation model [15, 26, 2] having only the relation and tuple constructors, thus producing a recursive schema with a tree structure. Abiteboul and Beeri [1] consider a set constructor adding to the algebra a powerset operator to reach the expressive power of the domain independent calculus.

In our case, besides considering different kinds of constructors, the main original contributions are related to the need to deal with object identity, and, at the same time, to preserve the capability of handling value based complex structures without identity. These features are not considered in most of the previous proposals which referred to strictly value based models.

Object identifiers are treated as special atomic values that cannot be directly accessed, and have to be preserved in the relational operations. This had to be taken into account in the definition of the semantics of the operators, and demanded for additional operators to convert objects into structured values and vice versa.

The extension of the operators of the relational algebra is attained by introducing in the conditions a set of *navigational* operators to move through complex data structures, e.g. to take components from structures, to extract elements from sequences, and to transform data types.

Nest and *Unnest* operators are defined as in [18, 25]. These allow respectively the grouping of several tuples introducing a set constructor over an attribute or group of attributes, and the distribution of the elements of a set attribute over a set of tuples. Similar operators are also provided for the tuple constructor. We may point out that, due to the introduction of the navigational operators in the conditions, the role of the Nest and Unnest is mainly restricted to the restructuring of data.

Further primitives are defined to transform objects or their components into tuples and vice versa. The latter operation generates new objects and requires object invention.

4 Conditions on structured types

To extend the operators of relational algebra we need to extend the definition of the conditions to deal with structured data types.

At the user (external) level, a program is expressed in a rule-based declarative language, and then a compiler is in charge to determine its structure and stratification, by using standard techniques (see for example [28, 11]), and eventually to rewrite it as an equivalent LOA program.

A *LOA program* is a sequence of *blocks*. Each block can be a single LOA algebraic equation or a *fixpoint block*. Each fixpoint block consists of a FIXPOINT operator applied to a set of LOA algebraic equations. These blocks are sequentially evaluated by the LOA processor, which maintains the mapping between the object schema and the relational schema, and translates the object algebra expressions into relational algebra with conditioned loops constructs to compute the *stratified fixpoint* of the block, as shown, for example, in [4, 9].

A more detailed description of a LOA program is the following. A *LOA program* P is a sequence of *fixpoint blocks*: $P = \langle \mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_m \rangle$, where the generic block B_k is either a single equation R_k or it is built up by a FIXPOINT operator applied to a set of equations: $\langle \textsc{fixpoint}\{R_{k,1}, R_{k,2}, \ldots, R_{k,n_k}\}\rangle$. Each equation $R_{k,i}$ has the form $V_{k,i} = \mathcal{E}_{k,i}$ where the left hand side is a typed variable and the right hand side $\mathcal{E}_{k,i}$ is a *LOA expression* on typed variables and constants. The type of each variable can be a class type or a relation type, and a constant is a class or relation in the knowledge base.

A variable $V_{h,j}$ can occur in the right hand side of a rule $R_{k,i} : V_{k,i} = \mathcal{E}_{k,i}$ if it occurs in the left hand side of some other equation either in the same block or in a previous blocks, that is if $h \leq k$ (thus including the variable $V_{k,i}$ itself). In the case where a block B_k is constituted by a single equation $R_k : V_k = \mathcal{E}_k$, then $\mathcal{E}_k$ can only contain references to variables in the previous blocks. Moreover, since the program is stratified, if a variable $V_{h,j}$ appears as a second term of a difference operator (hence coming from a negated predicate) it can not be defined in the same block, but only in the previous ones.

The structure of a LOA program can be summarized as follows:

$$
\begin{array}{l}
\langle\ \textsc{fixpoint}\{V_{1,1} = \mathcal{E}_{1,1}(V_{1,1}, \ldots, V_{1,n_1}) \\
\qquad\qquad V_{1,2} = \mathcal{E}_{1,2}(V_{1,1}, \ldots, V_{1,n_1}) \\
\qquad\qquad \ldots \\
\qquad\qquad V_{1,n_1} = \mathcal{E}_{1,n_1}(V_{1,1}, \ldots, V_{1,n_1})\} \\
\quad \textsc{fixpoint}\{V_{2,1} = \mathcal{E}_{2,1}(V_{1,1}, \ldots, V_{1,n_1}, V_{2,1}, \ldots, V_{2,n_2}) \\
\qquad\qquad \ldots \\
\qquad\qquad V_{2,n_2} = \mathcal{E}_{2,n_2}(V_{1,1}, \ldots, V_{1,n_1}, V_{2,1}, \ldots, V_{2,n_2})\} \\
\quad \ldots\ldots \\
\quad V_k = \mathcal{E}_k(V_{1,1}, \ldots, V_{1,n_1}, \cdots, V_{k-1,1}, \ldots, V_{k-1,n_{k-1}}) \\
\quad \ldots\ldots \\
\quad \textsc{fixpoint}\{V_{m,1} = \mathcal{E}_{m,1}(V_{1,1}, \ldots, V_{1,n_1}, \cdots, V_k, \cdots, V_{m,1}, \ldots, V_{m,n_m}) \\
\qquad\qquad \ldots \\
\qquad\qquad V_{m,n_m} = \mathcal{E}_{m,n_m}(V_{1,1}, \ldots, V_{1,n_1}, \cdots, V_k, \cdots, V_{m,1}, \ldots, V_{m,n_m})\}\ \rangle
\end{array}
$$

In the execution phase each block is evaluated by the LOA processor in sequence according the *stratified fixpoint*. A simple description of the inflationary fixpoint and a comparison with other fixpoint computations in the case of the flat relational model is given in [12] and a more detailed presentation can be found in [11]. Not much is known about fixpoint computations in the case of a more complex data model (see, for example [4, 9]).

Figure 2: Type definitions for the sample schema

Figure 1: A sample schema

- If τ is a *tuple* value type with value set $\mathbf{V}$, and $\mathcal{C}$ is a class name, then $\omega = [\mathcal{C}, \tau]$ is an *object type*, and the corresponding value set is $\mathbf{V}_\omega = \Omega \times \mathbf{V}$.

Each relation is defined on a value type and each class is defined on an object type. More precisely, as we shall see below, there is a one to one correspondence between object types and classes.

We may now introduce the notion of *refinement* as a partial order relationship in the set of types, according to the following definition.

A type θ (either an object type or a value type) is a *refinement* of a type θ' (in symbols $\theta \preceq \theta'$) if and only if one of the following condition holds:

- $\theta = \theta'$;
- $\theta = \omega = [\mathcal{C}, \tau]$, $\theta' = \omega' = [\mathcal{C}', \tau']$, with $\tau \preceq \tau'$;
- $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_{k+p} : \theta_{k+p})$, $\tau' = (\mathcal{A}_1 : \theta'_1, \ldots, \mathcal{A}_k : \theta'_k)$, with $\theta_i \preceq \theta'_i$, for $1 \leq i \leq k$;
- $\tau = \{\theta\}$, $\tau' = \{\theta'\}$, with $\theta \preceq \theta'$;
- $\tau = \langle\theta\rangle$, $\tau' = \langle\theta'\rangle$, with $\theta \preceq \theta'$;

In the logidata model, the ISA relationship defines both an intensional and an extensional constraint.

Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be two classes and Ω_1, Ω_2 respectively be the set of the identifiers of the objects in the two classes. The constraint $\mathcal{C}_1$ ISA $\mathcal{C}_2$ states that the type ω_1 must be a refinement of ω_2 and moreover that $\Omega_1 \subseteq \Omega_2$. Object types that have a common supertype in the ISA hierarchy are in a special relationship and are said to be *compatible* types.

In Figures 1 and 2 a sample schema is presented with the related type definitions.

3 Basic features of the Logidata Object Algebra

User programs are written in the LOGIDATA+ language, that is a rule based language on an object oriented data model, supporting identity. A stratified form of negation is allowed in the user external language. If the program is stratified, it is possible to rewrite it as a set of formulas which are *monotonic*, i.e. a repeated application of the formulas increases the collection of data stored in the variables. In the case of a complex data model with structure operators and object invention this does not guarantee that a fixpoint exists. We define a *stratified semantics* for our FIXPOINT operator.

Indeed it would be possible to adopt an *inflationary* semantics: this approach, used in [4, 9], is strictly more expressive than the stratified semantics [12] and has been proved to coincide with the least fixpoint [16] in the case of finite structures. Nevertheless, in the case of a complex data model, this is not a sufficient condition for a fixpoint of a set of equations to exist [4] and the computation may not terminate. The study of satisfactory solutions for this problem deserves further research effort.

A very high level description of the overall processing of user programs follows, clarifying the role of LOA.

2 The Data Model

LOA provides set-oriented operators to manipulate collections of complex objects and structured values, which are defined according to the LOGIDATA+ model [7], whose main features are the following:

- it allows the definition of types, functions, and two kinds of data collections: the *class* (based on object identity), and the *relation* (value based);
- data are basically structured with the *tuple*, *set*, and *sequence* constructors;
- an *isa* hierarchy with multiple inheritance can be defined in the set of classes.

Given a finite set of *domains* $\mathbf{D}_1, \ldots, \mathbf{D}_D$ with *domain names* $\mathcal{D}_1, \ldots, \mathcal{D}_D$, a countable domain of *object identifiers* Ω, and a countable set of *attribute names* $\mathcal{A}_1, \mathcal{A}_2, \ldots$, we refer to a database schema composed by the following elements:

- A finite set of *types names*, or shortly *types*, $\theta_1, \ldots, \theta_\Theta$, and the corresponding *type definitions*.
- A finite set of *classes* $\mathbf{C}_1, \ldots, \mathbf{C}_C$ with names $\mathcal{C}_1, \ldots, \mathcal{C}_C$.
- A finite set of *relations* $\mathbf{R}_1, \ldots, \mathbf{R}_R$ with names $\mathcal{R}_1, \ldots, \mathcal{R}_R$.
- An ISA hierarchy between the classes.

The domain Ω contains the object identifiers that are associated to the objects, each one having an identifier which is unique in all the database. Classes are collections of objects of the same type, and relations are collections of structured values.

Type definitions allow to build structured types from the basic types associated to the domains, and, for *object types*, to add the identity. A *value set*, i.e. the set of all possible values, is associated to each type. More formally:

- A *type* θ is either a *value type* τ or an *object type* ω.
- A domain name $\mathcal{D}_i$ is a *value type* and the corresponding value set is $\mathbf{D}_i$.
- If $\theta_1, \ldots, \theta_n$ are types with value sets $\mathbf{V}_1, \ldots, \mathbf{V}_n$, then $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_n : \theta_n)$ defines a *tuple type* τ, with value set $\mathbf{V}_\tau = \mathbf{V}_1 \times \ldots \times \mathbf{V}_n$. Round brackets denote the *tuple* constructor.
- If θ is a type and $\mathbf{V}$ is the corresponding value set then $\tau = \{\theta\}$ defines a *set type* τ with value set $\mathbf{V}_\tau = \text{PART}(\mathbf{V})$, i.e. the powerset of V. Curly brackets denote the *set* constructor.
- If θ is a type and $\mathbf{V}$ is the corresponding value set then $\tau = \langle\theta\rangle$ defines a *sequence type* τ with value set $\mathbf{V}_\tau = \text{SEQ}(\mathbf{V})$, i.e. the set of all the sequences over V. Angle brackets denote the *sequence* constructor.

to possible settlements, and furthermore different hardware/software environments can be tested for prototyping. Moreover the existence of such an intermediate language (in turn subjected to possible extensions) might point out the consequences, in terms of practical performances, of different choices at user language level and implementation of lower level primitives.

Several approaches for an internal language can be considered, characterizing the underlying data management support, consisting in all cases of a traditional environment enriched by several features in order to deal with the expressive power of the external language:

- a rule based language on a flat (relational) data model, that is a sort of datalog, supporting tuple identity, and limited forms of negation;
- a procedural algebraic language on an object oriented data model, in which additional features must be embodied to handle recursive queries.

We remark that in the case of a flat relational data model it has been shown that the first two approaches — generalized versions of datalog and relational algebra — may lead to the same expressive power, provided that suitable enhancements and restrictions are defined such as, for example, a limited form of negation (or complementation) and the absence of functions producing new data values not included in the active domain of the data base. Moreover this can be achieved with reasonable (polynomial) computational complexity (see, for example [12]). Unfortunately in the case of a more complex data model, this is not true, and the problem can achieve an arbitrary complexity [22, 17] and even become undecidable [4].

In this paper we focus on the algebraic approach, consisting in a suitable extension of the relational algebra to support the features of the LOGIDATA data model and external language. The Logidata Object Algebra (*LOA*) extends the relational algebra in several aspects, incorporating and extending some ideas and features from heterogeneous sources [6, 18, 15, 26, 2, 25, 27]:

- the semantics of the classical operators of the relational algebra are suitably redefined in order to deal with the richer object oriented data model;
- more operators are included to build and navigate complex data structures;
- procedural features are included, such as the FIXPOINT operator, in order to deal with recursive programs.

In this paper the problem of extracting data from an existing database is considered, and more precisely the problem of expressing by an algebraic language possibly recursive queries in an object oriented environment. This extends the approach that the authors presented in [23] to deal with recursion.

The rest of this paper is organized as follows. In the next section the basic elements of the formal definition of the LOGIDATA+ model are summarized. In section 3 an overview of the features of LOA is described together a basic semantics of LOA. In section 4 we introduce the language used to express complex conditions inside the operators of the extended relational algebra. These operators are discussed in section 5, and in the following one we introduce the structure and conversion operators to explicitly handle transformations between objects and values.

1 Introduction

In the last decade the relational model as proposed by Codd [14] has been widely adopted for standard applications, because of its simple and uniform structure and of the larger deal of independence between the logical and physical level.

However the relational approach has proved to be not satisfactory for several non-conventional applications, as CAD, CASE, office automation, multimedia databases and knowledge bases. In all these cases the flat structure of the model makes the representation of complex and structured data clumsy.

Several proposals have been made to extend the relational model, both overcoming the restrictions imposed by First Normal Form [18, 15, 26, 25], and introducing the concept of *object identity* [20, 21, 19, 4, 27]. For such extended models several authors have presented query and manipulation languages based on calculus [8], on algebra [2, 15, 1] and logic [3].

Dealing with the LOGIDATA language and model requires to support a rich and expressive environment, that is a rule based language in conjunction with an object oriented data model.

In this context the conceptual distance between the user (*external*) language and the implementation level, both in terms of data structure and language constructs, can be tackled by properly choosing some intermediate level in query processing (the *internal* language), in which user transactions have to be compiled before proceeding in the execution process on physical data. Of course such an internal language must have at least the same expressive power as the external language.

Our choice is to adopt an intermediate set-oriented manipulation language that will play in our prototype a role equivalent to that of the relational algebra in a relational system, that is providing a suitable framework to develop efficient access methods for data handling and storage, and powerful query processing strategies.

This approach makes especially sense when the object oriented system is aimed at traditional database applications, where the typical transaction involves large sets of objects. For this class of applications indeed the relational systems perform already quite well, and the motivation for using the object oriented model lies mostly in making more natural both the user language and the design phase, and, in general, in having a more direct representation of the real world in the schema. Therefore in such a context the relational model can still be used as an internal level of representation, by mapping the object schema into a relational schema.

According to these ideas a prototype system has been developed within LOGIDATA+, a national project funded by the Italian National Research Council, that has the goal of integrating logic programming and extended relational databases, with complex data types.

In general an approach to database query and management based on an intermediate language may have the disadvantage that it makes less effective the optimization process, which must be broken in two distinct phases, but has a number of advantages. A good level of modularity can be achieved in the design of the global architecture: this has the obvious consequence to separate different kinds of problems (source code interpretation and data processing) in different modules and, more interestingly, it allows a very fast and cheap experimentation of different solutions. This seems to be a basic requirement in an experimental context in which both the user language and the data model can be subject

The LOGIDATA+ Object Algebra*

Umberto Nanni† Silvio Salza‡ Mario Terranova‡

TR-92-006

February 1992

Abstract

In this paper we present the *LOGIDATA+ Object Algebra* (LOA), an algebra for complex objects which has been developed within the *LOGIDATA* project funded by the Italian National Research Council (CNR). LOGIDATA+ is intended to provide a rule based language on a data model with structured data types, object identity and sharing. LOA is a set-oriented manipulation language which was conceived as an internal language for a prototype system supporting such a rich environment. More precisely it is supposed that user programs, expressed in the external high level language have to be compiled into *LOA programs*, and then processed by an underlying *Object Manager*. In other words LOA is supposed to provide a suitable formal framework to develop efficient access to mass storage and powerful query optimization strategies. The algebra refers to a data model that includes structured data types and object identity, thus allowing both classes of objects and value-based relations.

LOA must deal with a rule based language with possible recursive programs with limited forms of negation. LOA programs explicitly include a FIXPOINT operator over a set of equations. Another original feature of the algebra is the ability to cope with object identity, and, at the same time, to preserve the capability of handling value based complex structures without identity. This is obtained by extending the semantics of classical operators of the relational algebra to deal with the LOGIDATA data structures, and by introducing additional operators, such as type conversion and oid invention. The paper also briefly discusses some implementation issues of the FIXPOINT operator and the other algebraic primitives.

*This work is partially supported by the project "Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council (CNR).

†Dipartimento di Matematica Pura e Applicata, University of L'Aquila, via Vetoio, Coppito - 67010 - L'Aquila, Italy. Currently visiting the International Computer Science Institute, 1947 Center St., Berkeley, CA 94704, U.S.A.

‡Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185, Rome, Italy.