Approaches to Deductive Object-Oriented Databases *

Alvaro A A Fernandes, Norman W Paton, M Howard Williams and Andrew Bowles†

Department of Computing and Electrical Engineering, Heriot-Watt University,

Riccarton, Edinburgh EH14 4AS, Scotland

This paper is concerned with the problem of combining deductive and object-oriented features to produce a deductive object-oriented database system which is comparable to those currently available under the relational view of data modelling not only in its functionality but also in the techniques employed in its construction and use. Under this assumption, we highlight the kinds of issues that have to be tackled for a similar research strategy to produce comparable results. We motivate our terms of comparison, characterise three broad approaches to deductive object-oriented databases and introduce the notion of *language convergence* to help in the characterisation of some shortcomings that have been perceived in them. Three proposals that have come to light in the past three years are looked into in some detail, insofar as they exemplify some of the positions in the space of choices we have defined. The main contribution of the paper is towards a characterisation of the language convergence property of deductive database languages which has a key role in addressing critiques of the deductive and object-oriented database research enterprise. We assume a basic familiarity with notions from deductive databases and from object-oriented databases.

Keywords. Deductive databases; object-oriented databases; data modelling; Datalog; Prolog.

1 Introduction

Scope and Goals

Deductive object-oriented databases (DOODs) constitute the most recent research area falling within the intersection of logic and databases. They reflect a growing awareness of researchers to the needs of both database designers and users for a representation tool which is semantically richer than that provided by relational databases (RDBs), as well as to the widely acknowledged benefits of using logic as a tool for the formalisation of both the static and the dynamic aspects of databases. In the past few years the research activity centred on DOODs has grown at an impressive rate — not only has a biannual international conference taking them as their theme already been held twice [23, 38], the number of publications focusing on the more general goal of integrating ideas from logic programming and the object-oriented world is also mounting rapidly. It is clear to us that any attempt at surveying the area in-depth and comprehensively would be a major undertaking, and we wish to stress we have *not* attempted to provide any such survey in the limited space of this paper. The proposals we will come to discuss, and the viewpoint from which we will come to consider them, are subordinated to our main objective in this paper, which we proceed to describe.

The research strategy adopted in the development of deductive relational databases (DRDBs) could be characterised as that of conceiving a RDB as a store of facts within a theorem-proving framework. We assume that this research strategy is equally worth pursuing if the data modelling primitive used to represent facts is the notion of an object instead of that of a relation. One of our main objectives in this paper is to characterise the language convergence property of DRDBs that allowed them to be developed so successfully in such a comparatively short time and to argue that it is also crucial for the success of DOODs.

To meet this goal, we associate with each of the kinds of database under consideration, viz. relational and object-oriented, a language space defined by abstracting away from details in particular proposals. More specifically, we fix as the unique source of the space, the logic that underlies every language derived from it. As a result, in a given language space, points are languages, while what we call *approaches* are development paths in the same space. We then proceed to consider briefly some proposals on the basis of the approaches they adopt. Note, however, that such language spaces cannot contain, or be provided for, every existing proposal merely as a consequence of their being the product of a particular abstraction. In other words, some proposals would simply not be captured by the abstraction principle we have used.

The language spaces are instrumental in allowing us to characterise the importance of a property of DRDBs which we

*In Information and Software Technology 34(12), December 1992, pages 787-803

†Now at the Department of Artificial Intelligence, University of Edinburgh

take to have been crucial to their success. Our claim is that if the research strategy used for DRDBs is to be emulated for DOODs then this property must be clearly acknowledged as a necessary one. Hence, the main contribution of this paper is to characterise this property of deductive database languages obeying the topology of a language space determined by the abstraction principle described above. We call this property *language convergence*. It is defined on nodes in the language space. Naturally, several possible development paths, exist. We claim that some of these paths lead to languages that converge to a lesser degree, or less strictly, than others, thus providing a means for comparing them. We are then able to point out that at least one line of criticism raised against the DOOD enterprise can be addressed by the separate formal definition of a data model taking objects as its modelling primitive. With clearly understood languages for the definition and the manipulation of objects, the degree of convergence which can be achieved is much greater, and so is the ability of the resulting DOOD to cater for crucial object-oriented notions.

It is not to be implied that the proposals we discuss are the only ones worth mentioning or that these, and others we only mention in passing or not at all, would not come to be judged as fully successful under different initial assumptions.

Antecedents, Problems and Perspectives

A *deductive database* allows users to deduce facts concerning data stored in the database using the logic-programming paradigm of computation. DRDBs, i.e. those built on the assumption that the underlying data model is relational [21], are well-understood by now and some implementations are reasonably well-known (for these, see [13], especially chapters 5 and 12). The transition from conventional RDBs to deductive ones was relatively smooth and natural because the relational data model has a simple, independent and well-understood expression in terms of first-order predicate logic (FOPL) [27, 50]. The idea of coupling a RDB to a relation-based logic-programming language (such as Prolog) can be motivated either by the enhanced expressiveness which deduction brings to RDBs, or by the efficiently managed, shared, persistent store which RDBs bring to deductive systems.

The scenario is altogether different if an *object-oriented database* (OODB) is required instead. Wegner [53] defines an *object* to be "a collection of operations that share a state". This definition gives precedence to the behavioural view of objects, viz. from the users' point of view, what matters about objects is the behaviour they display and not their underlying structure. In this paper we assume that OODBs must not undermine these expectations by their understandable concentration on representational issues. In other words, in order to benefit from data independence users have long been advised to shift the responsibility for managing their data requirements from application programs to enterprise-wide databases under some management system. We assume that, if those application programs were written in an object-oriented programming language, the move to an object-oriented database should not prevent users from perceiving the objects stored in the OODB primarily by the behaviour they display. Note that this raises important data modelling questions concerning the dynamic aspects of data that in relational data modelling only arise in a comparatively much simpler form.

OODBs are perceived to be most desirable in non-business applications, of which the classical examples are those supporting design activities or graphic data manipulation [8, 25, 58]. The object-oriented view of information systems relies crucially on four features [7, 8]:

1. **object identity** - in OODBs two entities in the real world can be distinguished in spite of having identical values for all their properties, as long as the database came to be aware of them through distinguishable events, referred to as object creation;
2. **classes with encapsulation** - in OODBs a varied and extensible type system is available to organise knowledge about sets of real-world entities, this knowledge is of two kinds:
   (a) *about structure* - defining which properties comprise the state of instances of the class,
   (b) *about behaviour* - defining which operations (or methods, as they are usually called) are granted exclusive rights of altering the state of instances of the class, a notion known as *encapsulation*;
3. **inheritance** - in OODBs classes are organised in a hierarchy so that a subclass inherits the structure and behaviour of each of its superclasses;
4. **late binding** - in OODBs the same method can be applied to instances of different classes, and the interpretation of a method name used in a request can only be carried out when the system knows the class of the object to which the request is addressed, i.e. at run time.

Clearly, there is a conceptual mismatch between logic-programming languages such as Prolog and the notion of *object* we are assuming to be the relevant one. This raises a series of difficult problems that must be tackled for DOODs to be seen as a step forward. Any solution seems to involve compromise involving either or both the database and the deductive engines. In fact, addressing the question of whether either of them can subsume the other and stand alone is a main concern of this paper. Some of the problems in this respect were raised by Ullman in [52]. They can be described as follows:

1. **Identity of entities obtained by different proofs** - a naive (Ullman prefers to say 'serious') approach to object identity could lead to the creation of unnecessary objects. The example given by Ullman is based on the recursive definition of *path* from the *arcs* of a graph:

   ```
   path(X,Y) <- arc(X,Y).
   path(X,Y) <- path(X,Z) & arc(Z,Y).
   ```

   A ground instance of $path(X,Y)$ is taken to mean that there exists a path that starts in node X and ends in node Y in the graph determined by the *arc* relation. For instance $path(a,b)$ asserts that there exists a path from a to b in the relation *arc*. Given two nodes a and b the different paths that may exist between them are treated as a single one in a DRDB, because in a value-based interpretation all facts of the form $path(a,b)$ are identical. In an identity-based interpretation one might (Ullman prefers to say 'must') create for each different proof of the fact that there exists a path from a to b a different entity, i.e. a distinct identity with value $path(a,b)$.

   One flaw in this example is that it depends on an ambiguous use of what the predicate symbol 'path' is used for. As stated in Ullman's argument, the predicate symbol 'path' is assumed, in the DRDB case, to denote (i.e. to be interpreted by) a binary relation with both arguments being points. What we know if the sentence $path(a,b)$ evaluates to 'true' is that a and b are connected. In the DOOD case, however, Ullman assumes that the predicate symbol 'path' must denote a ternary relation, since it is clear that what we now know if the sentence $path(a,b)$ evaluates to 'true' is that a and b are connected by a sequence of intermediate points. This brings out the fact that, in Ullman's argument, *path* is denoting different things.

2. **Dynamic typing in ad-hoc queries** - an ad-hoc query declaratively phrased by a user in a high-level language may have as result entities of unforeseen type. The example given by Ullman is based on joining two relations, R and S with schemes (A,B) and (B,C) respectively, to produce a third relation T with scheme (A,B,C). In a DRDB this could be written as the rule:

   ```
   t(A,B,C) <- r(A,B) & s(B,C).
   ```

   Note that all the operations of relational algebra are well-defined on the new relation T and its elements. In an OODB it is not easy to see how a new type can be left undefined, or be defined on the fly if one takes the latter to mean defining which operations are to be allowed on instances of the newly created type as well.

3. **Optimising declarative queries** - since OODBs support progressively higher levels of abstraction by defining ever more general types accompanied by their admissible operations, it could be argued that to cater for the declarativeness and ad-hoc nature of a join such as $R \bowtie S$, an OODB could have a defined class `relation` on instances of which the expected methods, such as `join`, are admissible and interpreted in the expected way. This would allow one to write expressions such as `R join T`. Ullman's reply is that this does not suffice for optimising ad-hoc queries, insofar as the properties of the method `join` that would allow an optimiser to handle properly the expression `R join T` are not known in any equivalent manner to that in which the properties of the relational algebra operator $\bowtie$ are.

Finally, a point not mentioned by Ullman in [52] is related to **unification** [51], which lies at the heart of the logic-programming approach to computation. In DRDBs, unification is over terms whose functors, i.e. relation names, are free (i.e. uninterpreted) and whose arguments are free 0-ary function symbols, i.e. atomic values of attribute domains, and in this case linear unification algorithms exist. In OODBS, arguments can be set-valued, for instance, and set unification is known to be NP-complete, again barring naive approaches.

These issues must be tackled if the logic-programming and object-oriented paradigms can be combined in a deductive database framework under similar assumptions to those that were applied in the combination of logic-programming and relational modelling. In this paper we argue that the role of a data model in providing the basis for addressing most of the above issues has not been fully recognised. In particular, we argue that by leaving certain data manipulation aspects of the framework partially undefined leads some influential proposals without clear answers to the above issues, and we have in mind, e.g., the definition of the abstract operations supported by objects in general, i.e. seen as a general-purpose modelling primitive

The remainder of the paper is structured as follows. In section 2 a brief description of the strategies that have been used in recent years to design DRDBs is given. We then characterise Datalog as the converged language from Prolog and the relational sublanguages. This allows us to depict the relational language space within which the motivation and constraints influencing the design of Datalog are reflected. We point out that extensions to Datalog exist and that each raises important problems for the theorem-proving aspects of the framework. In section 3 we concentrate on the language convergence problem under DOODs. The lack of clear candidates as the source languages to converge is characterised. We propose that three approaches, among others, suggest themselves more prominently, and indicate how they configure different language spaces. In sections 3.1, 3.2 and 3.3 we look in some detail into proposals that we see as adopting each of these three approaches. We summarise our analysis in section 4 by comparing the approaches and indicating the shape that answers to the perceived problems in combining deductive database with OODBs might take. Finally, in section 5 we draw our conclusions by explaining the importance of an explicit data modelling formalism in addressing Ullman's criticisms.

2 Strategies in DRDBs

Two main strategies have been used to build DRDBs, viz. *coupling* and what we propose to call *language convergence*. Briefly, the coupling strategy assumes that an existing deductive system and an existing database management system (DBMS) will be made to interact by the mediation of two interposed software components, usually called the deductive interface and the database interface. The language-convergence strategy seeks instead to produce a more integrated deductive database system, by redesigning the interaction between the deductive engine and the database engine to the point in which they become indistinguishable for all practical purposes, insofar as they operate on expressions of a single, converged language.

The Coupling Strategy

This strategy consists of using a general, computationally-complete deductive language, usually Prolog, and providing its main-memory resident data store with a connection to a full-fledged DBMS in as transparent a way as possible. The connection can be thought of as comprising an escape valve and a pipeline. Thus, whenever the deductive component is short of facts in the main-memory data store, the valve is activated and the pipeline pipes in from the mass-storage device a new supply of facts so that deduction can proceed.

The coupling strategy can be further refined by distinguishing between *loose* and *tight* coupling. In a loose-coupling approach, the logic-programming system accesses the database by generating calls via the latter's own query language, which is the normal route for application programs. This is safe (because preserving integrity remains entirely the responsibility of the DBMS) but prone to inefficiency. In a tight-coupling approach, the logic-programming system accesses the database by generating calls directly to the latter's access methods. This is more efficient but less safe (as, in a sense, the DBMS is being bypassed with respect to integrity checking). It should be remarked that others (e.g. [47]) use the above notions differently and call tightly-coupled what we prefer to call converged.

The attractiveness of coupling comes from its generality with respect to both the system that is used to perform deduction and the DBMS that is used to store the data involved. Although most Prolog-coupling systems have used RDBs, arguably most results would still hold for non-relational systems. Note that with coupling, performance hinges on adequate engineering, and the interest of major vendors is evidence for this. Nevertheless, some problems remain which are inherent in the coupling strategy itself. Firstly, the engines cannot be tuned by taking into account each other's characteristics; secondly, there is an overhead involved in managing the interaction between the components of such a multicomponent architecture (the impedance mismatch, for instance, plays a large part in determining this overhead); and, finally, there is a trade-off between safety and efficiency that emerges, respectively, in the loose and tight variants of coupling. It should be stressed that more recent integrated systems based upon Prolog, such as [11], have engineered solutions to overcome many of these problems in a RDB setting.

The Language Convergence Strategy

Language-convergence helps in overcoming the problems inherent in coupling and is attractive also from the point of view that it is less dependent on the actual engineering of a particular implementation. The price is paid in the form of a loss of generality. This comes about precisely because the point of making the languages converge is to produce a finely-tuned formalism in which storage issues are as adequately and transparently treated from the perspectives of both the deductive and the database engines. Intuitively, this process results in a formalism that is a specialisation of, and thus subsumed by, both the original engines.

In other words, attention is given to a triple of languages: the *logic-programming language* used in the deductive system, the *data definition language* (DDL) and the *data manipulation language* (DML) used in the database system.

The language convergence strategy consists in fine-tuning these three languages to one another, so that, for instance, the benefit of being able to express in a single declarative language queries, integrity constraints and derived data, is available to users. Consider Datalog, arguably the best-known example of such a converged language.

Datalog as a Converged Language

Datalog [13] is a language for deductive databases which was designed to be converged with respect to extensional RDBs. This was done by constraining the expressiveness of Prolog within the limits imposed by relational-database semantics and by adjusting the computational paradigm of Prolog to that of the relational sublanguages.

It is important to make explicit the assumption that the deductive language and the database languages converge only if, from the point of view of logic, the *terms* of the languages involved (which relate to their model theory) and their respective *evaluation strategies* (which relate to their proof theory, or computation paradigm) become as similar as possible. Note that, from a somewhat different perspective, the failure to meet these two requirements is another way of characterising the *impedance mismatch*, in both its facets, viz. the typing-system mismatch and the computational-paradigm mismatch. The languages will be said to converge if there is a formal account of their respective interpretation taking the same set of entities as their interpretation domain, which is thus *shared* in a formal sense. Note that in ordinary Horn-clause logic programs a clause such as

```
employee(12345,john,15000,sales)
```

is part of the domain of interpretation of programs using a predicate `employee`. The clause is used as a fact to compute the truth value of closed formulae in which that predicate is used. Now, in a RDB, the tuple `(12345,john,15000,sales)` is part of the domain of interpretation of queries using a relation

```
employee(id,name,salary,dept).
```

In this sense, one can say that there is a formal account of how logic programs and relational queries are interpreted such that a shared domain of interpretation exists. This observation can be used to inform the design of an integrated language. In the integrated language, correspondences are expected to be so direct that the mediation role played by the interfaces in the coupling strategy is significantly reduced or even ceases to be necessary. This is how efficiency is maximised, the impedance mismatch is minimised and guaranteeing database safety becomes an achievable goal.

Now, if the source languages are Prolog and the relational sublanguages, the convergence that is being sought implies stripping out of Prolog the extralogical features that impair the declarativeness and the safeness of programs, minimising the data-structuring features available, and adjusting the clause-at-a-time computational mode of Prolog to the set-at-a-time mode used with relations. For this reason, Datalog, which is the targeted language for this triple, has *no function symbols*, *no order-sensitivity*, *no extralogical-predicates* (such as the infamous **cut** and updating predicates), and *no unsafe notions* (such as negation in rules and built-in comparison predicates). It is *set-oriented* and (usually) adopts a *breadth-first search strategy.* The language that results is referred to as pure Datalog, to distinguish it from several proposed extensions.

Pure Datalog is a safe, recursive query language for RDBs not only capable of ranging over the usual extensional database (i.e. ground facts about the predicate to which the relation corresponds), but also of defining an intensional database expressed by means of Horn clauses interpreted as rules. As such, it subsumes in an important way the relational sublanguages from which languages such as SQL and QUEL originated.

Of course, the ease with which Prolog and RDBs can be made to converge in a language such as Datalog depends not only on implementing the adaptations mentioned above but, much more crucially, on keeping and, as is in fact the case, even tightening the logical equivalence of concepts, so that the model theory for the language becomes a close account of the admissible instances of the database, and the proof theory for the language becomes a close account of what is involved in answering queries and retrieving data from the database. In the relational case this is possible because every element in the triple ⟨*Prolog, relational DDL, relational algebra*⟩ can be formally defined from a common origin, viz. FOPL.

The language space we have discussed in the preceding paragraphs is depicted in figure 1. Note that two branches emerge from the source in FOPL: a *lower* one which is governed by a theorem-proving approach to computation; and an *upper* one which determines a data model by its sublanguages for the definition and manipulation of the formalism's modelling primitive, and ends up embodied in particular database management systems. The arrows indicate that mappings can be formally given between the languages concerned. In particular, note that in the case of figure 1 such mappings bridge between the two branches in both directions, reflecting the strict convergence achieved by the languages in the space.

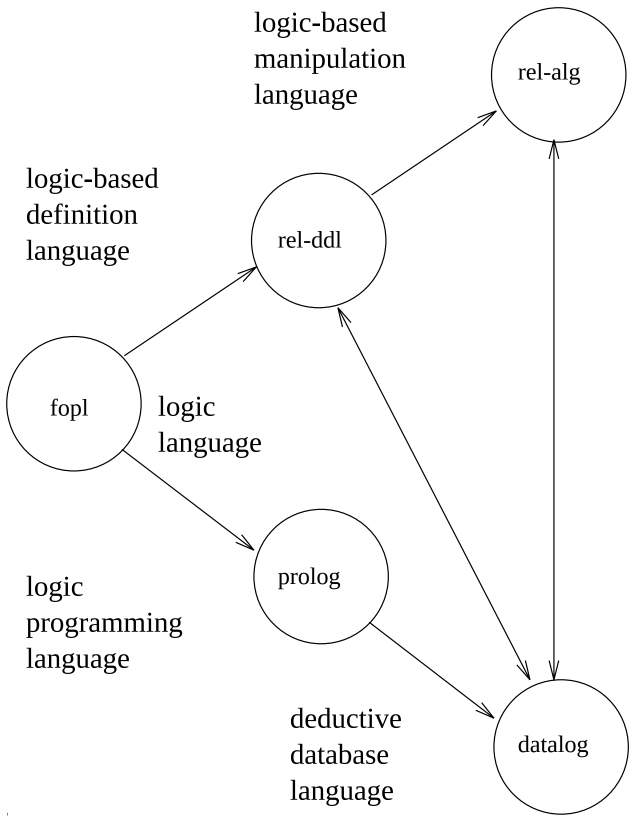

Figure 1: The Relational Language Space

Datalog Extensions

The qualifier *pure* is applied to Datalog because that language can be seen as falling into one extreme position in the language-convergence approach, viz. the extreme in which a predefined database component is given the highest priority. But we can conceive of several languages falling in positions between this and the opposite extreme. As we move from pure Datalog towards the opposite extreme, and allow the deductive component also to have a greater influence, we find that a set of extended languages have already been proposed. These give rise to an expressiveness hierarchy, as pure Datalog is progressively enhanced with features such as built-in predicates for equality and comparison, negation, and so on.

These features do introduce problems of their own. Predicates for equality and comparison can lead to unsafe programs which return infinite answers unless safety conditions are placed on clauses containing that kind of predicates. Negated predicates can also be unsafe. In the case of negation, however, solutions tend to be much more complex (again, see [13] for an extensive treatment of these problems). However, note that these extensions introduce problems to the theorem-proving aspect of a deductive database, rather than to its data modelling aspect. Due to this fact they resurface in every deductive database built under the theory of logic programming, irrespective of whether an OODB is being assumed instead of a RDB. Further, note that the same observation applies to the unification problems introduced by the use of interpreted functors and of complex terms to represent the complex types found in OODBs. Since our main goal in this paper is to consider how greater attention to data modelling issues addresses problems such as the ones raised by Ullman (see section 1), a detailed account of the nature and proposed solutions to those theorem-proving problems, relevant as they are to DOODs, lies beyond the scope of this paper.

In what follows when we use the term 'Datalog' we mean an adequately extended version of pure Datalog, handling arithmetic, comparison predicates, negation and complex terms, and possibly, but not necessarily, declaratively expressed updates. A well-known exemplar of these extended versions of Datalog is $\mathcal{LDL}$, fully described in [47].

3 Approaches to DOODs

There is some agreement on the superiority of the language-convergence strategy for the design of a well-founded and easy-to-use DOOD. The problem lies on which configuration of languages to start from. Our claim in this paper is that the origins of the problem can be largely traced back to the lack of an object-oriented data model (OODM) which can be compared to Codd's relational model (RDM) in terms of the latter's logical foundations, simplicity, and intuitive appeal. The design of such a data model has proved an elusive goal, in the sense that despite some very important attempts (e.g., [10, 42], among others), no formal model of objects as a data modelling primitive has yet been granted widespread acceptance.

One immediate consequence of this is that proposals for DOODs must either specify their departing DDL and DML or leave them to be inferred or, in the extreme case, wholly ignore them. Bearing in mind the consensual qualities of the RDM, it can easily be seen that this whole initial-selection process becomes a much more complicated task in the case of OODBs. In other words, DRDB researchers could take for granted that in the triple ⟨*logic-programming language, DDL, DML*⟩ the second and third elements, far from being variables, were in fact ground to those of the RDM, and only the choice of the logic language needed be an issue at all. Having a predicate-based, hence relational-based, language such as Prolog at hand, even this last choice point was much constrained. DOOD researchers, however, must specify all three elements insofar as Prolog is no longer such an obvious choice and database sublanguages for OODBs are barely formalised in most cases. Roughly, the only element in the language space on which there is agreement is in taking some first-order logic as the source of the space, though this does not necessarily mean that this logic is to be based on predicates as they are classically construed.

For the purposes of the remaining exposition, we concentrate on three possible approaches that have been used to define a converged language for DOODs. These will be referred to as: the *OO-Datalog* approach, the *OO-Prolog* approach and the *OO-logic* approach. We stress once more that our choice of proposals to exemplify each of the approaches was guided only by their adequacy in illustrating our argument. Under this assumption, citing many other comparable proposals would just have been redundant.

3.1 The OO-Datalog Approach

We apply this label to proposals that depart from Datalog, i.e. which assume that there is no significant conceptual conflict which could prevent a further-extended Datalog from hosting an OODM. We will take as our example a family of three languages: COL (Complex Object Language) [2], IQL (Identity Query Language) [5], and the nameless language recently proposed in [1], which, for convenience, we shall call **KL** (**K**yoto **L**anguage, for the place where the language was disclosed). Together they subsume Datalog by extending it with certain object-oriented features and updates.

Main Features

This family of languages can be characterised by the adoption of Datalog as a platform from which a deductive object-oriented database language can be derived in a stepwise manner. The three languages mentioned above are steps that progressively try to incorporate more notions from the object-oriented paradigm, without compromising the goal of having the relational languages as particularisations of the extended one.

This evolution can be briefly summarised as follows. COL was proposed as an extension of (pure) Datalog with complex typed-terms. It also had other design objectives which are of less interest in our present context. COL uses a fixpoint semantics. It is, however, a value-based language and does not cater for the essential object-oriented notion of identity. IQL extended COL with object identity, thus providing the essential element for the modelling of inheritance and for a richer type system. IQL supports classes as sets of objects identifiers, as well as relations as sets of values. Object identity also plays a part in improving the declarative handling of updates and set terms. IQL was seen as the operational part of a model whose structural aspects were object-oriented. Note, however, that what is taken as the operational part of a model in this context is not nearly as complex as suggested by Wegner's definition (see above, section 1). Nevertheless, IQL represented an important step ahead in the direction of a robust and well-founded DOOD.

KL, the most recent member of the family takes previous achievements further by providing the language with methods, thus bringing the resulting language closer to the object-oriented paradigm. Recalling the four crucial features of OODBs mentioned in section 1, viz. object identity, classes with encapsulation, inheritance and late binding, KL is lacking only in fully catering for some aspects of object-oriented behaviour modelling which play a part in allowing users to view object-orientation as being tantamount to interaction by means of message-exchanging [53]. In particular, although KL has methods and method inheritance, the interplay of encapsulation and late binding is not addressed. The presentation that follows concentrates on the language we are calling KL [1], with reference being made to the other two languages [2, 5] where necessary for clarity of exposition.

Definitional Aspects

The notions of object-identity, classes and types are basic, with classes being names for sets of object-identifiers whose structure is defined by some type. Two constructors are used, for sets and for tuples. The distinction between schema- and instance-level concepts is retained. This is one important property of this group of languages which distinguishes them from the Stony Brook family which we will come to consider in section 3.3.

A schema is declared using a type language. Examples of statements are:

```
class person:
      equal student or employee
      and   [name:string];

class student:
      isa   person;

class employee:
      isa   person
      and   [supervisor:senioremployee];

class senioremployee:
      isa   employee;

class dept: [dname:string];
...
```

From statements such as these it is possible to catch a glimpse of the definitional expressiveness of the language. The keyword `equal` is used to define a class as a *generalisation* (a person is either a student or an employee, i.e. no objects are declared to be persons: the extension of the class `person` is derived by taking the union of the extensions of its subclasses `student` and `employee`). The keyword `isa` is used to define a class as a *specialisation* (a senior employee is an employee, i.e. some objects are declared to be employees, but the extension of class `employee` also contains those declared as senior employees). Expressions enclosed in square brackets define tuple-types, and colons associate attribute names to types. Structural inheritance proceeds from such declarations.

One aspect that is not evident from the above is that the notion of *relation*, a set of values (as opposed to object-identifiers), all of the same type, is also supported. This indicates the concern with keeping two versions of the model, viz. the main identity-based version, and a value-based one, which motivates several OO-Datalog proposals, and this family in particular.

Manipulative Aspects

Methods are modelled as functions obeying typing constraints, and in the spirit of COL can either be stored extensionally or derived from rules. Since the COL proposal, stress has been placed on the utility of the data-function notion in terms of allowing a switch between a relational and a functional view of the extensional and intensional database. In a predicate-based language, data functions can be characterised as relations constrained by a functional requirement, which the system is responsible for enforcing. Thus, just as atomic formulae can have their extension asserted as relations in the database, so do functional terms. It is claimed that this facility clears the way for a cleaner integration of functional and relational approaches to database systems [4].

Methods can be declared as follows:

```
method graduationdate:
       person -> <[university:string,
                   date:int]>;
```

This statement declares that `graduationdate` is a function from persons to lists of tuples having as first element a university of type `string` and as second element a date of type `int`. Methods can map into single- or set-values, the latter being taken to include lists as well.

Methods are defined on classes and inherited as expected. Overloading and overriding are possible by associating methods with more than one typing constraint in the method declaration and by propagating this under inheritance. However, it is not clear from [1] how the functionality associated with late binding is to be provided.

The language handles updates by extensions that are briefly described below. Beyond the notion of a program as a set of rules with a fixpoint semantics, which is said to constitute a *local level*, the language is further extended to what is called a *top level* by the use of explicit control on which rules are active and which are not, and then to an *external level* which provides extensibility by embedding it into a host language. Features that result from this extended language include:

Views and Updates - contrary to the norm in deductive databases, not only views (i.e. intensional predicates) but also database elements (i.e. extensional predicates) can appear in the heads of rules. The way this is handled is by explicitly declaring relation- and method-declarations in the schema to be views or not. This controls the interpretation of heads of rules, i.e. if a rule head is not a view, whenever its body is proven true for some instantiation this is interpreted as a request for inserting a new fact into the extensional database. Deletion is signaled by a negative literal in the head. Both facts and rules can be inserted and deleted.

Constructs for Explicit Control - programs can be structured with the aid of constructs such as **begin-end** blocks, **if** rules (execute once) and **while** rules (execute iteratively). The last two will help to define the notions of triggers and integrity constraints. Finally, procedures using local temporary relations and methods can be defined using a **let ... be** construct.

Persistence - schema elements must be declared to be **permanent**, as opposed to **temporary**, in order for them not to be discarded at the end of the session.

External Methods - methods can be written in host languages and referred to in rules.

The language also provides for the creation of object-identifiers (through variables that only appear in the head of a rule) and for their inclusion in a class. In particular, a new object need not be created if an existing one fulfills the specifications. As object-identifiers can be created without an initially assigned state, the language provides for the modification of the state of objects, as well as the modification of the value of methods (recall that methods can be extensionally stored as relations under a functional constraint). Finally, it is also possible to modify an object-identifier assignment, and thus remove an object-identifier from a class or transfer it from one class to another.

Deductive Aspects

The rule-based language uses constants, tuples, lists and sets as terms. As declared in the schema, class and relation names are terms, as are methods with properly typed arguments. Built-in operators provide for projection (denoting a field in a tuple), dereferencing (denoting the value of an object-identifier) and the classical *cons* operation on lists.

Literals are expressions built from properly typed terms using equality and membership predicates. Note that a predicate such as `worksfor(X,Y)` has the alternative notation `worksfor` ∋ `[X,Y]`, where the square brackets indicate a tuple term.

The database is thus a set of facts (i.e. ground literals) about class members, object values, relation tuples, and methods. Examples of these are, respectively:

```
employee(#1)

*(#1) = [john,phil]

worksfor(#1,#7)

graduationdate(#1) = <[yale,1965],[mit,1968]>
```

In these examples, object-identifiers have been prefixed by `#`, although this is not a requirement of the language. The dereferencing operator is `*`. List terms are enclosed in angle brackets, tuples in square brackets, sets in curly brackets.

Rules have a positive literal as head and a list of positive and negative literals in the body. As derived facts are obtained from rule heads the particular form these take is determined by the kinds of terms that are stored in the database (see examples above). Since all terms are typed, the logic is technically many-sorted.

The membership predicate is used in derived methods to express facts concerning set-valued objects, while the equality predicate achieves the same purpose for single-valued ones. Thus suppose that a method is declared as:

```
method friends: student -> {person};
```

then the rule `friends(X)` ∋ `Y <- teaches(Y,X)` allows the derivation of a fact about a set-valued object, viz. the set of all people Y who are friends of X because Y teaches X, i.e. because `teaches` ∋ `[Y,X]` holds. Correspondingly, suppose a method is declared as:

```
method managedby: employee -> string
```

then `managedby(X) = *(X.supervisor) <-` is an (unconditional) rule allowing the derivation of a fact about a single-valued object, viz. the name of the senior employee of an employee X. Note that if o is an object-identifier, $o.A$ uses the projection operator . to denote the attribute A of o.

The semantics for the language uses stratification to handle negation, the existence of set-valued terms, and also the non-monotonicity that results from the overriding of methods. Fixpoint computation on the resulting strata is used to compute the facts that can be derived from the program.

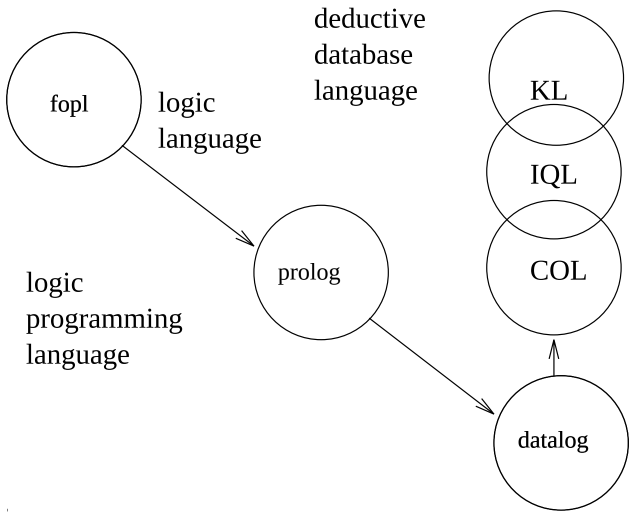

Figure 2: The COL/IQL/KL Language Space

Summary

The language space determined by the COL/IQL/KL family of languages we have discussed is depicted in figure 2. Contrast this with figure 1. Here there is no definition of a data model, as spelled out in its sublanguages, that is separate from the deductive database language. We acknowledge this fact by arguing that KL is not strictly converged, in the sense of being related to preexisting object-oriented database languages that are independently defined and independently used. This analysis helps in understanding that the underlying data model which Datalog inherits from the RDM via FOPL is deeemed to be extensible to an object-oriented one by features progressively added to Datalog in COL, IQL and KL. KL was designed to be a stand-alone language, with no provision for simple mappings to any existing OODB management system (OODBMS). Hence, it assumes the burden of catering for all perceived improvements that users expect from adopting the object-oriented paradigm. As an aside, note that a fully-fledged logic-programming language such as Prolog is dispensable, though one could argue that KL is the equivalent to Prolog in this scenario, in its concern with practicality.

3.2 OO-Prolog

We apply this label to proposals that depart from (full) Prolog, again under the assumption that no significant conceptual conflict arises from this. The OO-Prolog approach is represented by [26, 45, 48], and we will take [48] as our example.

Much work [22, 46, 56, 57] has also been done on extending Prolog with object-oriented notions. The motivations, strategies, techniques and goals are defined more from a programming-language point of view than from a database-language one. These proposals have in common their assumption that the syntactic machinery of Prolog suffices to express several object-oriented notions. They are of less interest in the context of this paper since their concern with data modelling issues is subsidiary. Because of this, important problems from a database perspective tend to be either disregarded as side issues or treated in a simplistic way.

Proposals here, from both the programming- and the database-language point of view, tend to be harder to classify on the basis of the language-convergence notion. In this approach it is often the case that the reliance on one of the elements of the language-triple at the expense of the others results in a heavily imbalanced language.

Main Features

Throughout the 1980s, in parallel with the largely theoretical research into DRDBs summarised in section 2, there was a separate implementation-oriented activity which looked to develop database systems based upon Prolog. This branch of research gave rise to a number of implementations of coupled or more fully integrated Prolog/Database systems [30]. Such systems could subsequently be used to implement database systems in which Prolog was used to implement such components as the data model, alternative query languages and optimisers [29, 31, 54]. At the same time, researchers from outwith the database community were proposing various object-oriented programming systems implemented in Prolog [26, 45, 56], in which the flat Prolog clause base was structured using object-oriented mechanisms, while Prolog was retained as a manipulation language. It is from this broad, experimental environment that OO-Prolog systems and OODBs implemented in Prolog have emerged.

For OO-Prolog systems, the triple $\langle$*logic-programming language, DDL, DML*$\rangle$ can be considered to be partially instantiated as $\langle$*Prolog, x, Extended-Prolog*$\rangle$. While the DML is ostensibly Prolog, there are usually a number of additional operators for performing such tasks as sending messages and updating structures. The data model varies considerably from proposal to proposal, with concepts such as encapsulation, classification and identity supported in a variety of different ways in different systems. The uninstantiated DDL in the triple above is a sign of this variety.

Definitional Aspects

This section uses the syntax of ADAM [48] to illustrate some typical data modelling features. The following definitions specify the schema of a simple parts database:

```
:- new([course,[
        slot(slot_tuple(code,global,single,
                            total,string)),
        slot(slot_tuple(level,global,single,
                        optional,integer))
    ]]) => class.

:- new([person,[
        slot(slot_tuple(cname,global,single,
                             total,string)),
slot(slot_tuple(sname,global,single,
      total,string))
    ]]) => class.

:- new([employee,[
        is_a([person]),
        slot(slot_tuple(phone,global,single,
                          optional,integer)),
        slot(slot_tuple(teaches,global,set,
                           optional,course))
    ]]) => class.

:- new([student,[
        is_a([person]),
        slot(slot_tuple(year,global,single,
                            total,integer)),
        slot(slot_tuple(takes,global,set,
                           optional,course))
    ]]) => class.
```

The message-sending operator `=>` sends the message specified by the first operand to the object which is the second operand. Each slot in ADAM is described by a number of facets, which specify its name, visibility, cardinality, status and type [48]. The statements for class creation given above are message-sends, as in ADAM all classes are themselves objects [49]. Instances of the new classes are in turn created by sending messages to the newly created classes:

```
:- new([SiD,[
        cname(['Fred']),
        sname(['Flintstone'])
   ]]) => student.
```

In this example, the Prolog variable `SiD` is unified with the system-generated object-identifier of the new *primitive_part*. Features of this specific model include multiple inheritance, encapsulation and run-time type checking.

Manipulative Aspects

Behaviour can be associated with classes by defining methods which are attached to a class either when it is created using *new*, or by the method *put_slot*. The following program attaches the method *num_courses* to the class *student*, which can be used to find the number of courses taken by a *student*.

```
:- put_method([(num_courses(global,single,
                       [],integer,[Num]) :-
        findall(C,
            get_takes(C) => self,
            CList),
        length(CList,Num)
    )]) => student.
```

Queries can subsequently be written which retrieve information from the database. For example, the following program writes out the *cname* and number of sub-parts of all *students* in the system. The method *get* retrieves the instances of a class, while the method *get_cname* retrieves the value from the *cname* slot of the object to which it is sent:

```
:- ( get(S) => student,
     get_cname(Cname) => S,
     num_courses(Num) => S,
     write('Name,' has ',Num,' courses'),nl,
     fail
   ; true
   ).
```

Deductive Aspects

The examples given above are of a specific OO-Prolog system, and the variety of such systems in the literature [26, 45, 46, 56] makes it difficult to extrapolate from an example system towards general conclusions for the family. We note, however, that the code in the examples above leans very heavily upon a procedural rather than a declarative interpretation of Prolog, in which objects have an encapsulated state which changes over time as the result of message-sends. The Prolog resolution theorem prover is used to evaluate both top-level programs and methods, where messages are sent either to retrieve information from the object-oriented store, to deduce properties defined using methods, or to change the state of the database.

In ADAM the distinction between classes and instances is preserved by forcing every object to conform to the structure defined in the class of which it is an instance. In certain other OO-Prolog systems [46, 56], this distinction is removed, and all new objects are *is-a* linked to prototypes whose properties they inherit.

A further characteristic of this family of systems is that an individual OO-Prolog is defined as a program (written in Prolog) which has some specific functionality. Thus, for example, support for inheritance is not specified within the fixpoint semantics of a language (as in [1]), or as part of the underlying logic (as in [34]), but rather by using an algorithmic approach. Further, issues such as safety are rarely mentioned by the proponents of OO-Prolog systems, and such systems inherit their lack of safety from the language upon which they are built. Indeed, the notion of objects which have a modifiable state generally means that many of the extra-logical facilities of Prolog are used extensively within the implementation of OO-Prolog systems.

Finally, the distinction made within Datalog between an intensional database (i.e. a set of clauses in the logic language) and an extensional database (i.e. a set of positive ground unit clauses) has no place in this approach. Deduction becomes a less precise notion.

Summary

The language space determined by proposals we label as OO-Prolog is much harder to depict. In figure 3 we offer no more than a general picture. As is the case of the COL/IQL/KL family, here also we find no separate definition of an OODM. However, while KL could be said to be converged, albeit in the less strict sense we have indicated when discussing the language space in figure 2, languages resulting from the OO-Prolog approach tend to be fully-fledged

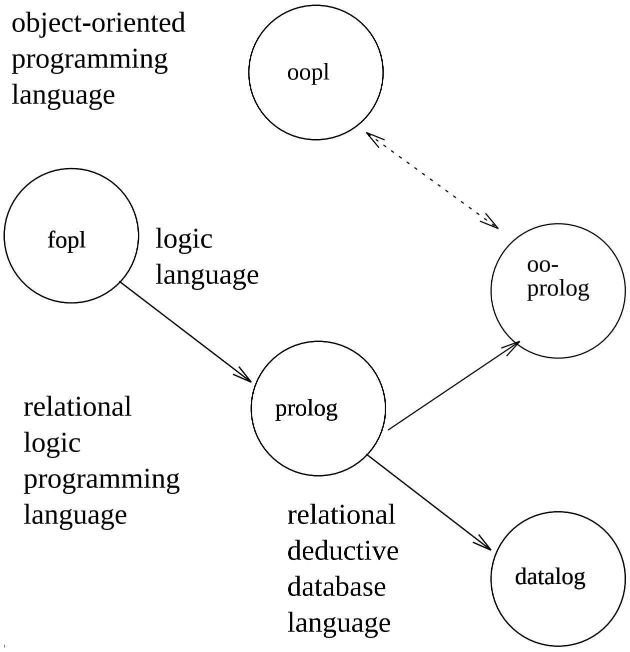

Figure 3: The OO-Prolog Language Space

logic-programming languages. In terms of language convergence, it can be argued that OO-Prolog systems are a short-cut which by-pass the language-convergence issue by using extra-logical features to introduce identity into a language which is essentially value-based. As such, the resulting systems retain both the virtues and vices of full Prolog. In fact, an OO-Prolog language is best seen, as is Prolog, as a programming language for applications which include, but are not limited to, those requiring a database, and in the case in hand, an object-oriented one.

In this sense of being unconstrainedly expressive, OO-Prolog proposals tend to be equivalent to object-oriented programming languages (OOPL) in general, hence it is easier for them to cater for the behavioural view of objects that we are assuming to be desirable for users. However, OO-Prolog languages also tend to be stand-alone, just as KL was seen to be. This is to say that OO-Prolog languages cannot in general be formally mapped into other languages, e.g. the OOPL of some existing OODBMS. (In figure 3 we indicate by dotted arrows the nonexistence of such mappings.) The difference to the OO-Datalog approach resides in the fact that while languages belonging to the latter are harnessed by logical strictness, OO-Prolog languages in general are not.

3.3 OO-Logic

We apply this label to proposals whose departure points are logics that alter in nontrivial ways the FOPL foundations which underlie Prolog and Datalog. This implies a degree of doubt as to whether these languages can serve as a platform for building DOODs. The rationale behind this approach is the belief that one must redesign the deductive language from a more fundamental point. We will take our examples from the a family of three languages developed at Stony Brook whose common origin is the now superseded Maier's O-logic [44]. As Maier himself acknowledged the existence of problems in his original proposal, this immediately prompted other researchers to improve the original O-logic. Thus, C-logic [20] is an improvement of the original O-logic. The proposal in [36] corrects and extends Maier's original proposal. This revised version of O-logic is such that C-logic is a proper subset of it. Finally, F-logic [33, 34] extends the revised version of O-logic.

Main Features

F-logic, the most recent member of this family of languages, is the outcome of a rethink on the kind of logic language one needs to deal with object-oriented features. One crucial question here is: will the language only refer to individuals or will it allow reference to be made to complex structures in which individuals can be seen to participate in the real world (e.g. sets)? Relational systems, for instance, choose the first alternative; OODBs choose the second. This means that OODBs are placed beyond the realm of classical first-order languages. This is relevant because a well-known result from mathematical logic states that it is not possible to provide a full-fledged higher-order logic system with a complete axiomatisation. To tackle the problems that result from that, the Stony Brook approach ultimately led to F-logic, a language whose expressions are not syntactically first-order but which admits an interpretation in first-order semantics. In other words, the model theory with which F-logic is equipped is such that all references are ultimately to individuals, rather than to structures they are perceived to be composing. Beeri [10] has argued that, while technically speaking a restriction on the expressiveness of the language, the technique used for this mixture of a higher-order syntax and a first-order semantics is probably adequate in the context of databases.

As we have discussed in section 2, Prolog and Datalog operate under assumptions about the static and dynamic aspects of the store of facts which the RDM amply realised. Under the Stony Brook view of the problem, switching from the RDM to an OODM means that new assumptions come into play, and that to cater for them as transparently one must have at hand at least a second-order syntax. In other words, an OODM brings with it a nontrivial reformulation of the static and dynamic aspects characterising the store of facts on top of which the deductive database language is now to operate. Thus, the latter must emerge from a more foundational look into the underlying logic. A problem that stands in the way of such strategy is the fact that no proposed OODM has a clear formulation in logic, hence there is no clear picture of what shape such a reformulation of the store of facts will end up having from a logical point of view.

Some of the technical novelties incorporated in the C-/O-/F-logic family are discussed in [18], in the more general context of logic programming. In what follows we will concentrate on F-logic [33, 34], as it is by far the most expressive member of this class of logics. The version published in [33] will be the one used here as it is the most easily accessible. The revised version [34] contains changes which, though significant in several aspects, do not impact the discussion in this paper.

F-logic abolishes any significant distinction between schema and instance, or, to put it in other terms, between names and their denotations. By imposing a lattice structure on the universe of ground terms, this reification of intensional objects such as classes is used to derive a model-theoretic account of inheritance and a treatment of locally inconsistent information.

Another contribution of the Stony Brook approach is a better account of how new object identities are to be created as a result of rule evaluation. Skolemisation provides an existentially quantified interpretation to variables appearing in the head of a rule but not in its body. Then, when such a rule succeeds, this can be taken to be asserting the existence of an object whose identity is the denotation of the Skolem function. To illustrate this, consider the addition of a third argument to the `path` predicate used in section 1:

```
path(X,Y,i) <- arc(X,Y).
path(X,Y,o(I,Z)) <- path(X,Z,I) & arc(Z,Y).
```

Then, the third argument acts as a trace of the derivation of a `path(X,Y)` fact. As an example, suppose there are two paths from node `a` to node `c`, where the first is given by the tuple of nodes `<a,d>` and the second by `<a,b,c,d>`. Then the identity of the first path would be `i`, that of the second would be `o(o(i,b),c)`, where the second arguments at each level of nesting read from the deepest to the shallowest gives us the intermediate nodes in the path. Alongside with the proviso used in KL (see section 3.1) for creating new objects only if no existing object fulfills the specification, this Skolemisation technique for creating new object identities provides an appealing account for the problem of object creation in theorem-proving frameworks such as DOODs.

Definitional Aspects

The fundamental notion in F-logic is that of *id-term*. Id-terms denote individual objects in a classical way, thus a (complex) F-term can consist of attributes defined on sets of id-terms (a second-order syntactic construct) without jeopardising the first-orderness of the semantics. The notion of id-term can be seen as the F-logic account of object identity.

F-logic uses a domain that is organised in a lattice according to IS-A relationships. Classes (or more precisely, class-identifiers) and instances are equally placed as members of this domain. "This feature", it is claimed in [33], "is largely responsible for the fact that inheritance is naturally built into the semantics of F-logic, contrasting it to algorithmically defined inheritance in other approaches."

F-logic takes the general approach of giving full object status to all elements in the language. This leads to an unconstrained syntax in which symbols can appear in practically every syntactic role, and this, in turn, leads to variables being able to range over almost every conceptual notion.

Labels (the F-logic word for attribute-identifiers) are objects in their own right and, therefore, are typed as any other object. They can either be single-valued or set-valued. Complex terms can be built with indefinite depth of nesting.

A (complex) F-term can be exemplified by:

```
student:mary
```

```
  [name    -> 'Mary',
   age     -> 30,
   friends -> {bob,sally},
   works   -> dept:cs[dept-name -> 'CS']
  ]
```

where `student` is an id-term standing for a class-object, `mary` is an id-term standing for the object being defined, `name` and `age` are id-terms standing for single-valued labels, `friends` is an id-term standing for a set-valued label, `works` is an id-term standing for a relationship which the object being defined maintains with the object denoted by the F-term `dept:cs[dept-name -> 'CS']`. Note that `'Mary'`, `30`, `bob`, `sally`, and `'CS'` are also id-terms standing for objects in the domain.

The definitional power of F-logic is considerable, insofar as the model does not impose any important restrictions on what may be construed as an object. Work under this more foundational approach may provide a new basis for further developments because of its conceptual minimality, a property which proved to be a success factor for the RDM.

Manipulative Aspects

Inheritance in F-logic is built into the semantics of the language, in the sense that there exists a theorem stating that if a database satisfies a formula concerning properties of a given object then that formula is also satisfied for objects that, in the domain of interpretation, are hierarchically above (i.e. less general than, or more informative than) the former according to the lattice structure of the domain. This means that property inheritance is expressed directly in the structure used to assign meaning to formulae.

A powerful consequence of giving full object status to things like class-identifiers and attribute-identifiers is that objects that are traditionally considered to be schema-level elements, also exist at the data-level. They can be given attributes and be queried like any other objects. It is possible to construe a notion of the schema being dispersed in the statements of the language. In other words, interpreted in a database context, the statements of F-logic do not distinguish syntactically or semantically between schema-level and data-level information. It is up to users to do so when they wish. However, since updating the domain of interpretation can make the logic undecidable, a restriction is imposed whereby schema browsing is supported though schema evolution is not.

The notion of method is catered for in F-logic by taking advantage of variables ranging over id-terms. Consider an example from [33], where *graduation_date* is a method (i.e. a function from universities to years) defined by the following rule:

```
person:X[graduation_date(U) -> Y]  <=
  univ:U[alumni  ->
        {alumni_record:G[student -> person:X,
                         date -> year:Y]}]
```

This can be used to answer queries such as:

```
X[graduation_date(U) -> 1987]?

john[graduation_date(U) -> Y]?
```

The first query returns all graduates of all universities who have 1987 as a graduation date. The second query returns the universities and corresponding graduation dates for the individual `john`.

Note that this is possible because labels (a single-valued one in this case) are id-terms, and these do not have to be ground. Therefore, it is perfectly possible to construe a notion of methods as parameterised labels, or conversely, of attributes as parameterless methods. In [33, 34], a brief exemplification is given of how methods can be inherited and how overriding can be accounted for.

Deductive Aspects

One point worth noting is that predicate symbols are made available in F-logic only for the purposes of catering for a conventional-style syntax. However, predicate terms are notational variants of F-terms defined on class-identifiers, and these play the same role that predicate symbols play in a standard logic-language. Thus, for each predicate there is a corresponding F-term, obtained by mapping the predicate symbol into a class-identifier, supplying an id-term that is a function of both the class and the arguments, and devising labels for each argument. Using this construction procedure in reverse shows how F-logic can be reduced to a value-based framework. Notice that this is an example of the overall goals of the OO-logic approach, viz. a reworking of foundational aspects in such a way that a logic language can be blended with a language of objects in a more natural way, i.e. one based on a common core of conceptually equivalent notions.

In [34] a sound and complete proof procedure based on resolution is defined for the monotonic part of F-logic, extending the results published for the revised O-logic [36]. With respect to that proof procedure two remarks are important. First, the problem of set unification still seems to require a solution like the one adopted in $\mathcal{LDL}$ [47], viz. a rule containing a set term can be compiled into a logic-programming procedure in which the set term is transformed into a first-order term while the idempotence and commutativity properties of sets is captured by the structure of the rules in the procedure. Second, in classical logic-programming the control problem is greatly simplified by the fact that a single inference-rule, i.e. *resolution*, suffices for any proof (though it is optionally complemented by *factorisation* so as to reduce the number of disjuncts in a clause). By contrast, F-logic needs several additional rules. This has the effect of expanding the search space as there is now a choice point as to which rule the proof procedure should apply at any given stage. The claim is made that the consequences in terms of complexity are acceptable but the evidence presented for this does not dispense with the need for further investigation.

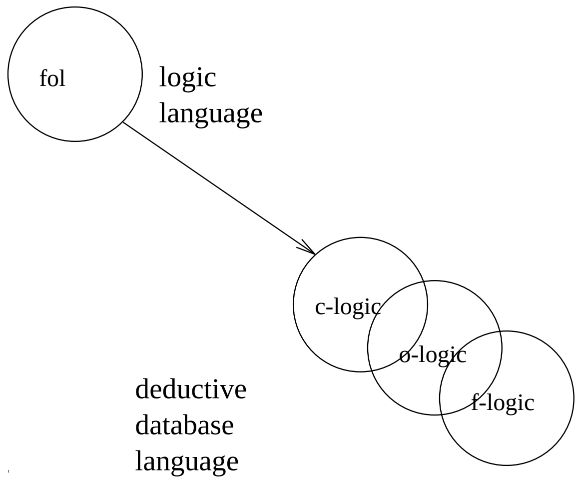

Figure 4: The C-/O-/F-logic Language Space

Summary

The language space determined by the C-/O-/F-logic family is depicted in figure 4. As one would have expected from a more foundational approach, this is a sparse space in comparison with those of the other approaches. More specifically, in comparison with the COL/IQL/KL language space in figure 2, it pushes back, so to speak, the point where the deductive language begins to be derived. Correspondingly, the C-/O-/F-logic family does not go as far in accounting for practical features of the resulting language, such as the updates and control structures KL provides.

The C-/O-/F-logic language space is, however, no different from the previous ones in not comprising the separate definition of an OODM. Again it is not possible to say unrestrictedly that the resulting deductive database language is converged. Hence, F-logic again assumes the burden of catering for all perceived improvements that users expect from adopting the object-oriented paradigm. It does go far in that: for instance, its account of overloading includes the case of methods defined in classes not related by inheritance, it provides schema browsing facilities, it manages to accommodate inheritance into the notion of semantic consequence. However, it is still not capable of modelling behaviour in the same abstracted way that users are accustomed to in object-oriented systems.

4 A General Summary

In this section we offer a brief evaluation of the proposals we have described. This is done in two stages: in the first we accept their own assumptions as reflected in the language spaces they generate; in the second we confront them with Ullman's criticisms [52] to argue that under his assumptions their responses may not be adequate. Finally, we indicate how adequate responses are possible.

We wish to remark that there is a wealth of research activity (e.g. [12, 43], among others) whose fruits would fit under one of the labels we used to discuss certain approaches to DOODs. The fact that we have not taken them as examples should not be taken as an implicit judgment of comparative merit. Other proposals are more difficult to classify within a general framework of DOODs. Among these, proposals exist to integrate sets in logic programming [24, 39, 40, 41], to handle non-first-normal-form relations in a deductive language [14], and to extend (pure) Datalog to handle nested relations [15, 16]. Others are, strictly speaking, programming languages: the SQL extension towards deduction and abstract data types (ADTs) proposed in [28] and the object-oriented programming language incorporating rules proposed in [55].

The COL/IQL/KL Approach

For these languages the main issue is to what extent object-oriented notions can be accounted for in terms of an underlying predicate-based view of the deductive language, while providing users with a message-passing paradigm of computation. It could be argued that an underlying first-order view is inescapable, since first-order logic is the only suitable formalism to take as a departure point. However, this may not preclude a more expressive syntax being devised which would cater for an easier-to-use language with no loss in well-foundedness.

As an example, note that the effect of the distinction between schema- and instance-level concepts is that proposals under this framework become less expressive than, e.g., F-logic, in the sense that this additional constraint reduces the kinds of admissible statements. Such a distinction may have a positive impact on implementation efforts, since a trade-off between expressiveness and efficiency is clearly present in logic-based database-modelling efforts.

KL is certainly an attractive proposal and closer to what one would expect from a database point of view than F-logic. Since it was conceived as a stand-alone language it tackles the issue of controlling execution, upon which database technology is dependent for implementing the notions of transaction, concurrency, and recovery, for instance. If DOODs are to be an evolution of DRDB, then KL gives a much clearer picture of this evolution than F-logic. The lack of multiple inheritance and of support for late binding are points that make KL less attractive from an object-oriented point of view.

The OO-Prolog Approach

Strictly, the OO-Prolog approach can be seen from two perspectives. In the first, the attempt is to incorporate OO features into the Prolog language. There is no implied restriction to simple database applications of a language such as Datalog. This is consistent with the trend towards database- and persistent-programming languages.

In the second, which we concentrated on, Prolog is regarded as the platform from which OODBs can be implemented. The main attraction of the OO-Prolog approach is the ease with which it may be implemented. In terms of a practical database system, access to a non-resident clause store is required. The main disadvantage is the lack of theoretical soundness of the full Prolog language, giving rise to problems with optimisation, tuple-based evaluation, order-sensitive programs, etc.

The main issue here is that no language-convergence is apparent, so that it is difficult to see how the seamless integration achieved in Datalog with respect to the relational model would be possible here with respect to the object-oriented paradigm.

The Stony Brook Approach

The Stony Brook approach has the greatest appeal in terms of providing a major breakthrough. Their main claim is that returning to foundations is the only way of guaranteeing that the blend of logic and OODBs will be as fruitful as it has been in the case of RDBs. This seems quite appealing, but more evidence must be produced that the conceptual mismatch between a classical predicate-logic and notions brought into play by object-orientation have actual consequences that cannot be solved by evolutionary approaches. Certain technical aspects of the F-logic proposal appear to substantiate the Stony Brook claim to some extent. For instance, extending the range of constructs that can be assigned as values of a logical variable results in an important increase in expressiveness. This, in turn, can be used to account for object-oriented notions hitherto untreated by other proposals in an integrated way. Also, languages have, besides a syntactic and a semantic dimension, a pragmatic one as well. The ease of use of a language, its conciseness and expressiveness, all are important factors determining the appeal it will have for users.

The main issue is essentially implementational, and at this stage there is no certainty as to whether the increase in expressiveness will not jeopardise the overall efficiency of systems. Overall the Stony Brook approach seems to be in a better position than the others to deliver a principled and pragmatically useful deductive database language, although practical systems are some way away.

5 Conclusions

The Independent Data Model Issue

We have noted that a common feature of the language spaces defined on each of the three approaches is the absence of an upper branch that is comparable to the one in figure 1. We claim that this upper branch, representing the independent definition of a data model from a common logical source, is indispensable if DOODs are to become useful in real-world situations. This could be argued for on pragmatic grounds alone, insofar as it seems much too ambitious to handle all the issues that arise in the context of enterprise-wide databases within a single linguistic framework, if the applications that are being catered for are complex enough to require the power of OODBs.

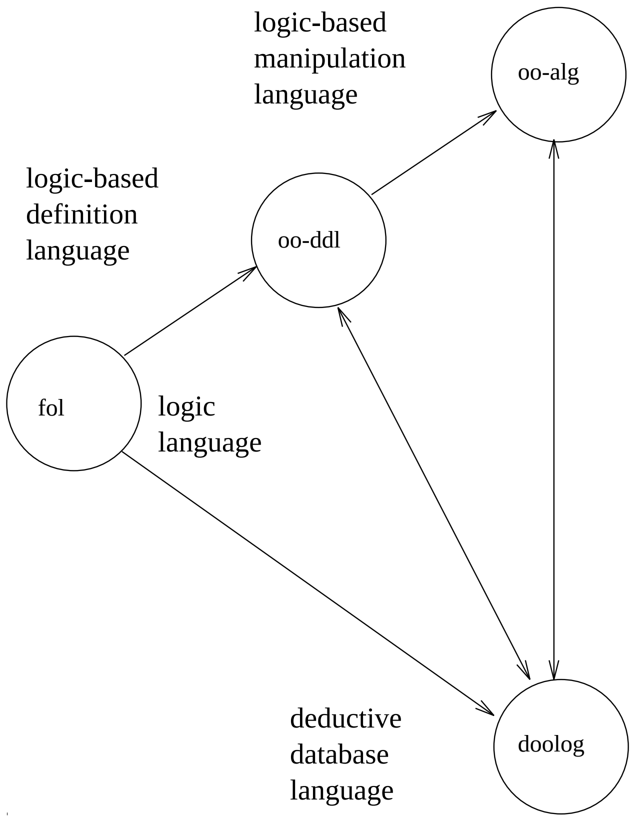

Figure 5: The Object-Oriented Language Space

Consider the object-oriented language space in figure 5. Apart from not containing an object-oriented equivalent to Prolog, it closely resembles the relational one in figure 1. It reflects the reliance of a deductive object-oriented language (which we are referring to as Doolog) on independent specifications of sublanguages for the definition and manipulation of objects as a modelling primitive.

Then, Doolog can be designed to be the converged language, strictly speaking, of a triple ⟨*FOL, OO-DDL, OO-Algebra*⟩. What we mean by *FOL* is some first-order language from which the two branches can spring. It could be classical FOPL, it could be some syntactically-adjusted version of it such as HiLog [18], or still, it could be some other, maybe new, logical framework or architecture. In any case, it *must* be capable of serving as the foundation of a Horn-clause resolution-based theorem-prover that is sound and complete. What we mean by *OO-DDL* is a language for defining objects, both structurally and behaviourally, with which application domains can be modelled. What we mean by an *OO-algebra* is an algebraic specification as ADTs of the objects yielded by the OO-DDL, i.e. the equational definition of a minimal set of operations that are well-defined on objects independently of any additional application-dependent typing and, naturally, of representational choices.

Different Proofs Implying Different Identities

We now confront each of the approaches we have discussed with Ullman's criticisms (see section 1). These are strongly centred on a view of deductive databases as the ideal ground for declaratively expressing ad-hoc queries. Among other things, such queries must be amenable to having their safeness concluded, to producing results which are immediately available for further manipulation, and to being optimised automatically. One could argue that this is a very limited view of deductive database technology, but it is by no means an irrelevant one.

Ullman's first point, about assigning different identities to entities derived by different proofs, is probably the easiest to answer, insofar as there is simply no reason why deductive object-oriented databases are compelled to create object identities blindly for each proof of each fact.

In most OO-Prologs, objects are created explicitly as opposed to being derived by means of rules. The COL/IQL/KL family already accommodates the option of *not* creating a new object identity if an existing object meets the specification, i.e. if it already has the value that would end up being associated with the new identity. Ullman tries to preempt this response by arguing that this is not to take object identity seriously, as he has chosen to phrase it. However, in a seminal analysis of this notion [32], a **merge** operation has been described that destroys a superfluous identity because the value associated with it is equal to that of another existing object. KL simply allows one not to create that superfluous identity in the first place, and can therefore be seen to be addressing an issue about taking object identities seriously. The COL/IQL/KL answer to Ullman's first point can be adopted in the two other approaches. As noted above, the creation of object identities suggested by the Skolemisation technique used in F-logic can be similarly adopted in the other two approaches. Combined they adequately address Ullman's first point.

Dynamic Typing of Answers to Ad-hoc Queries

Ullman's second point, about the problem of manipulating objects (in Wegner's sense, see section 1) which do not conform to any previously declared type, is harder for the approaches we have discussed to answer, although, as noted above, in many situations objects need not be implicitly created. DRDBs are not affected by this problem because they inherit from the RDM the algebra which results from taking relations as an ADT, and no strong domain-specific typing constraints are declarable in the RDM. In fact, no application-specific behaviour is specifiable in the RDM: operations on relations are the same across applications no matter how different they are in reality.

If one attaches a strict interpretation to the role of typing in OODBs, the question could well be found to be indeed very hard to answer: there seems to be no obvious way of deriving a new entity by means of rules and simultaneously endowing it with a collection of operations which define it in the eyes of users. However, the lack of distinction between schema-level and instance-level elements in F-logic suggests that facts about the schema (hence including those about application-specific behaviour) can be the subject of deduction on a par with facts about its instantiation. The problem then becomes how to constrain this power within bounds that are determined by the users' shared expectation about how every object, no matter its application-defined type, should behave

It is arguable that neither KL nor F-logic takes a view of objects that conforms to Wegner's definition which is the one assumed by Ullman. Because they do not abide by this behaviourally-determined definition of objects they largely preempt the question. If conforming to a type has no effect on behaviour and only impacts the structure of objects, methods can be thought of simply as attributes, i.e. functions from objects to values. Note that this question is tied to the question of encapsulation, a feature in which the proposals we described are clearly inadequate, with the exception of ADAM which has strong encapsulation.

We argue that an *OO-algebra* as we defined it above offers a suitable compromise, i.e. it results in DOODs that offer the same kind of functionality that DRDBs provide to users. Objects can still be behaviourally-determined in general, but objects that are derived by rules need only display that behaviour which is defined on any object whatsoever simply by virtue of their being an instance of a particular ADT. This is fully equivalent to the functionality offered by DRDBs.

Optimising Ad-hoc Queries

Ullman's third point, about the impossibility of optimising queries cast in terms of operations whose properties are not known to the optimiser program, can be addressed in similar fashion.

Again, by not facing up to the behaviourally-oriented definition of objects the proposals we have described can be said to not even acknowledge the problem. Again we argue that specifying objects as ADTs offers a suitable compromise. To use Ullman's example, if queries use methods such as `join`, the problem of assuring that its properties are known to a query optimiser could be reduced to the problem of giving it a semantics in terms of the basic operations defined on the object ADTs. This leads to a construal of methods as higher-level operations definable in terms of more basic ones. As the latter are algebraically defined by equations, a query optimiser has the means with which to conclude facts about equivalence and thus decide on the optimal formulation of a query.

Concluding Remarks

The DRDB research enterprise has been very successful. There is no reason not to try and use the same strategies used in it in the case of DOODs. However, criticisms have begun to appear to which the more prominent approaches to DOODs do not respond satisfactorily. In this paper, after describing the strategies used to build DRDBs, we noted that attempts of applying them to DOODS have overlooked the issue of an independently-defined but logically-compatible data model, mainly, we believe, because no such model has been proposed that is widely accepted as satisfactory. We argued that this definition, apart from being important in itself, is a necessary step in responding to a certain class of criticisms, because only then is it possible to characterise what we have described as a *converged language* for DOODs, of which Datalog is the equivalent in the DRDB case.

6 Acknowledgements

We would like to thank Dr. Keith G. Jeffery of Rutherford Appleton Laboratory for useful discussions on the subject of this paper. Our work is funded funded by the Science and Engineering Research Council through the IEATP programme, and their support is duly acknowledged. We would also like to thank Dr. J.M.P. Quinn of ICL and Mr. N. Smith of Ordnance Survey as the industrial partners in our present research project.

References

[1] Serge Abiteboul. Towards a Deductive Object-Oriented Database Language. *Data & Knowledge Engineering*, 5:263–287, 1990. Preliminary Version in [38], pages 453–472.

[2] Serge Abiteboul and Stéphane Grumbach. COL: A Language for Complex Objects Based on Recursive Rules. In *Proceedings of the Workshop on Database Programming Languages*, pages 253–276, Roskoff, 1987. Complete version in [4]. See also [3].

[3] Serge Abiteboul and Stéphane Grumbach. COL: A Logic-Based Language for Complex Objects. In Joachim W.Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Advances in Database Technology - EDBT'88, International Conference on Extending Database Technology*, LNCS 303, pages 271–293, Venice, Italy, March 1988. Springer-Verlag. ISBN 3-540-19074-0, ISBN 0-387-19074-0. See also [2, 4].

[4] Serge Abiteboul and Stéphane Grumbach. COL: A Logic-Based Language for Complex Objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, Frontier Series, chapter 21, pages 347–374. ACM Press/Addison-Wesley Publ. Co., New York, NY, 1990. Abstract in [2]. See also [3]. ISBN 0-201-50257-7.

[5] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 159–173, Portland,OR, June 1989. ACM Press. SIGMOD Record, 18(2), June 1989. This is a preliminary version of [6].

[6] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. Technical Report 1022, INRIA, April 1989. A preliminary version appears as [5].

[7] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley B. Zdonik. The Object-Oriented Database System Manifesto. In *[38]*, pages 223–240. 1990.

[8] François Bancilhon. Object-Oriented Database Systems. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 152–162, Austin,TX, March 1988. ACM Press.

[9] François Bancilhon and Peter Buneman, editors. *Advances in Database Programming Languages*. Frontier Series. ACM Press/Addison-Wesley Publ. Co., New York, NY, 1990. ISBN 0-201-50257-7.

[10] Catriel Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5:353–382, 1990. Also in [38], pages 405-430.

[11] Jorge Bocca. Megalog: A Platform for Developing Knowledge-Base Management Systems. In *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pages 374–380, Tokyo, 1991.

[12] Filippo Cacace, Stefano Ceri, S. Crespi-Reghizzi, Letizia Tanca, and Roberto Zicari. Integrating Object-Oriented Data Modeling with a Rule-based Programming Paradigm. In *Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data*, pages 225–236, Atlantic City, NJ, May 1990. ACM Press. SIGMOD Record, 19(2), June 1990.

[13] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990. ISBN 3-540-51728-6, ISBN 0-387-51728-6.

[14] Qiming Chen and Wesley W. Chu. HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases. In *[38]*, pages 431–452. 1990.

[15] Qiming Chen and Georges Gardarin. An Implementation Model for Reasoning with Complex Objects. Technical Report 793, INRIA, February 1988.

[16] Qiming Chen and Georges Gardarin. Nested Datalog: A Rule Language for Complex Objects. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL, 1988. ACM Press.

[17] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, October 1989. ISBN/ISSN 0-262-62064-2. Expanded Version in [18].

[18] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A Foundation for Higher-Order Logic Programming. Technical report, Department of Computer Science, State University of New York at Stony Brook (SUNY), 1989. Expanded Version of [19, 17].

[19] Weidong Chen, Michael Kifer, and David S. Warren. HiLog as a Platform for Database Languages (or Why Predicate Calculus Is Not Enough). In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 315–329, Gleneden Beach, OR, June 1989. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-072-8. Expanded Version in [18].

[20] Weidong Chen and David S. Warren. C-Logic of Complex Objects. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369–378, Philadelphia,PA, March 1989. ACM Press.

[21] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970. Also in CACM 26(1) January 1983 p. 64-69.

[22] Mukesh Dalal and Dipayan Gangopadhyay. OOLP: A Translation Approach to Object-Oriented Logic Programming. In *[38]*, pages 593–606. 1990.

[23] Claude Delobel, Michael Kifer, and Yoshifumi Masunaga. *Deductive and Object-Oriented Databases (Second International Conference DOOD'91, Munich).* Springer-Verlag, Berlin, 1991. ISBN 3-540-55015-1, ISBN 0-387-55015-1.

[24] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi. {log}: A Logic Programming Language with Finite Sets. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 111–124, Paris, July 1991. The MIT Press (Logic Programming Series).

[25] Elizabeth Fong, William Kent, Ken Moore, and Craig Thompson. X3/ SPARC/ DBSSG/ OODBTG Final Report, September 1991.

[26] K. Fukunaga and S. Hirose. An Experience with a Prolog-Based Object-Oriented Language. In *OOPSLA '86 Conference Proceedings Object-Oriented Programming: Systems, Languages and Applications*, pages 224–231, Portland,OR, September 1986. ACM Press.

[27] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[28] Georges Gardarin and Patrick Valduriez. ESQL2: An Extended SQL with Object and Deductive Capabilities. Technical Report 1185, INRIA, 1990.

[29] Peter M.D. Gray, David S. Moffat, and Norman W. Paton. A Prolog Interface to a Functional Data Model Database. In Joachim W.Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Advances in Database Technology - EDBT'88, International Conference on Extending Database Technology*, LNCS 303, pages 34–48, Venice, Italy, March 1988. Springer-Verlag. ISBN 3-540-19074-0, ISBN 0-387-19074-0.

[30] P.M.D. Gray and R.J. Lucas, editors. *Prolog and Databases: Implementations and New Directions.* Ellis Horwood, Chichester, 1988.

[31] M.H. Kay and P.J. Rivett. An Overview of the Raleigh Object-Oriented Database System. *ICL Technical Journal*, November, 1991.

[32] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *OOPSLA '86 Conference Proceedings Object-Oriented Programming: Systems, Languages and Applications*, pages 406–416, Portland,OR, September 1986. ACM Press. Also in [59], pages 37-46.

[33] Michael Kifer and Georg Lausen. F-logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 134–146, Portland,OR, June 1989. ACM Press. SIGMOD Record, 18(2), June 1989. A revised and extended version exists as [35].

[34] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14, Department of Computer Science, State University of New York at Stony Brook (SUNY), June 1990. Superseded by [35].

[35] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 93/06, Department of Computer Science, State University of New York at Stony Brook (SUNY), April 1993. Revised June 1993.

[36] Michael Kifer and James Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–383, Philadelphia,PA, March 1989. ACM Press. Preliminary Report of [37].

[37] Michael Kifer and James Wu. A Logic for Programming with Complex Objects. Technical report, Department of Computer Science, State University of New York at Stony Brook (SUNY), 1989.

[38] Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors. *Deductive and Object-Oriented Databases (First International Conference DOOD'89, Kyoto)*. Elsevier Science Press (North-Holland), Amsterdam, 1990. ISBN 0-444-88433-5.

[39] Gabriel M. Kuper. An Extension of LPS to Arbitrary Sets. Technical report, IBM, 1987.

[40] Gabriel M. Kuper. Logic Programming with Sets. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–20, San Diego, CA, March 1987. ACM Press. Revised version in *Journal of Computer and System Sciences*, 41(1):44-64, August 1990.

[41] Gabriel M. Kuper. On the Expressive Power of Logic Programming with Sets. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 10–14, Austin,TX, March 1988. ACM Press.

[42] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an Object-Oriented Data Model. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL, 1988. ACM Press. Also in [59], pages 227-236 and in [9], pages 257-276.

[43] Yanjun Lou and Z. Meral Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on the Management of Data*, pages 198–207, Denver, Co, May 1991. ACM Press. SIGMOD Record, 20(2), June 1991.

[44] David Maier. A Logic for Objects. Technical Report CS/E-86-012, Oregon Graduate Center, November 1986.

[45] J. Malenfant, G. Lapalme, and J. Vaucher. ObjVProlog: Metaclasses in Logic. In *Proceedings ECOOP*, pages 257–269, 1989.

[46] Frank G. McCabe. *Logic and Objects: Language, Application and Implementation*. PhD thesis, Imperial College of Science and Technology, London, November 1988.

[47] Shamim A. Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, Rockville, MD, 1989. ISBN 0-7167-8200-6.

[48] Norman W. Paton. ADAM: An Object-Oriented Database System Implemented in Prolog. In M. H. Williams, editor, *Proceedings of the Seventh British National Conference on Databases (BNCOD 7)*, pages 147–161, Edinburgh, July 1989. Cambridge University Press.

[49] Norman W. Paton and Oscar Díaz. Metaclasses in Object-Oriented Databases. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction(DS-4)*, pages 331–347, Windermere, July 1990. North-Holland. ISBN 0-444-88929-9.

[50] Raymond Reiter. Towards a Logical Reconstruction of Relational Database Theory. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–233. Springer-Verlag (Topics in Information System Series), New York, NY, 1984.

[51] Jőrg H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7(3 & 4):207–274, March/April 1989.

[52] Jeffrey D. Ullman. A Comparison Between Deductive and Object-Oriented Database Systems. In *[23]*, pages 263–277. 1991.

[53] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, August 1990. Expansion of Oct 4 OOPSLA-89 Keynote Talk.

[54] M.H. Williams, G. Chen, D. Ferbrache, P. Massey, S. Salvini, H. Taylor, and K.F. Wong. Prolog and Deductive Databases. *Knowledge-Based Systems*, 1(3):188–192, 1988.

[55] Limsoon Wong. Inference Rules in Object-Oriented Programming Systems. In *[38]*, pages 493–509. 1990.

[56] Carlo Zaniolo. Object-Oriented Programming in Prolog. In *IEEE International Symposium on Logic Programming*, pages 265–270, 1984.

[57] Carlo Zaniolo. Object Identity and Inheritance in Deductive Databases – An Evolutionary Approach. In *[38]*, pages 7–24. 1990.

[58] Stanley B. Zdonik and David Maier. Fundamentals of Object-Oriented Databases. In *[59]*, pages 1–32. 1990.

[59] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, CA, 1990. ISSN 1046-1698, ISBN 1-55860-000-0.