LDL: A Logic-Based Data-Language

Shalom Tsur
Carlo Zaniolo

Microelectronics and Computer Technology Corporation
9430 Research Boulevard
Austin, Texas, 78759

Abstract

In this paper we describe the considerations that led us to the design of LDL and provide an overview of the features of this language. LDL is designed to combine the flexibility of logic programming with the high performance of the relational database technology. The design offers an improved mode of control over the existing logic programming languages together with an enriched repertoire of data objects and constructs, including: sets, updates and negation. These advantages are realized by means of a compilation technology.

1. Introduction

In this paper we describe the considerations that led to the design of LDL and provide an overview of the features of this language. The motivation behind the design of this language stems from our desire to take advantage of the current developments in the area of logic programming as well as the recent advances in relational database systems.

The advantages of logic as a formal foundation of databases and knowledge management systems have long been recognized [Gall78, Gall84], and can be summarized as follows:

(1) It offers a high-level, uniform and consistent formalism for data, view and integrity-constraint definition that rest on a solid, theoretical foundation.

(2) it supports reasoning and inferential capabilities.

(3) It offers an adequate basis for the development of an even higher functionality in the future: support for temporal, non-monotonic and other forms of reasoning.

(4) It is general in the sense that it does not make an a priori distinction between stored-knowledge processing and general-purpose computing.

(5) It is amenable to parallel processing.

For all these potential benefits, logic has been used more often as a formal vehicle for specification than as a programming language in which to encode algorithms--predominantly as a consequence of the poor performance of early systems that employed logic for theorem-proving. Using Kowalski's celebrated equation: ***Algorithm = Logic + Control*** [Kowa79], we see that the problem of the efficient use of logic in computational tasks is tantamount to that of effective control. Whereas early theorem provers have failed in providing efficient control schemes, relational database theory and logic programming have recently provided efficient and widely used systems e.g., SQL/DS [IBM81] and [Quin85], which use two entirely different control paradigms. We compare these next.

Today's state of the art in logic programming is represented by PROLOG in its various manifestations [Clock84, Naish85]. We assume some familiarity with this language as well as the standard logic terminology. Because of the relative popularity of of PROLOG in this realm of computing we will use it in this paper for comparison with LDL. PROLOG is based on Horn-clause logic and a sequential execution-control model. Rules are searched and goals are expanded in the very order in which they are specified (SLD resolution). Thus, the responsibility for the efficient execution and termination of programs rests with the programmer: an improper ordering of the predicates or rules may result in poor performance or even in a non-terminating program. In addition, a number of extra-logical constructs (such as the *cut*) have been grafted to the language, turning it into an imperative, rather than a purely declarative language. This reduces PROLOG's amenability to parallel implementation and detracts from its ease of use and its data independence. The execution model of PROLOG, like other logic programming languages, simply assumes random access to the objects that it manipulates and relies on the virtual memory support of the underlying computer system when large-volume data is involved. Consequently, it employs a *tuple-at-time* nested-loop join strategy, which is not suitable for data access in secondary storage.

The query languages of relational databases are also based on logic -- the interpretation of a relational query amounts to a proof of satisfiability (in the model theoretic sense) see e.g., [Gall84a]. The control over their execution is however the responsibility of the system that, via query optimization and compilation techniques, ensures efficient performance over a wide range of storage structures and database demographies. A great deal of thought and ingenuity has been invested in the efficient processing of queries posed in database languages. The working assumption is that the volume of data to be manipulated is too large to be contained in the memory of a computer and, hence, that special techniques for secondary-memory data-access and update must be employed. Thus for instance, a relational system is not confined to nested-loop joins when implementing queries, but can take full advantage of the logically equivalent strategies for joining and accessing physical data.

Relational systems are superior to PROLOG with respect to ease of use, data independence, suitability for parallel processing and secondary storage access. However, the expressive power and functionality offered by database query languages is limited compared with the logic programming languages. Typically, they are designed around a *Data Model*. In other words, there is a distinction, at the application level, between *stored* or *base relations* and *derived relations* that can be obtained through the view mechanisms that these languages provide. The distinction between data-model and query language limits their expressive power: they do not support such constructs as recursion and general unification that entails the computation of closures and the use of complex structures (as opposed to just formatted records). Even more important, PROLOG, unlike relational languages is Turing-complete[1], and can be used as a general-purpose programming language. It is in fact being used so with great success in varied applications such as symbolic manipulation, rule-based expert system and natural language parsing. In contrast, relational query languages are often powerless to express complete applications, and are thus embedded in traditional programming languages. This method is known to have its drawbacks: primarily the "impedance mismatch" between the relational query and imperative languages and the lower programmer productivity when these languages are used.

The comparison between logic programming languages and database query languages provides the background and motivation for LDL. We want to combine the benefits of these two approaches by designing and supporting a logic-based query language that combines the power of PROLOG with the ease of use, the suitability for parallel processing and secondary storage management of the relational systems. The language will be purely declarative in nature in which the control over the execution will be the responsibility of the system. We expect that once realized, such systems will be used to support traditional database queries, data-intensive applications, knowledge based and expert applications, as well as deductive retrieval and inferential queries.

The design and implementation of this system, which we are currently undertaking, poses many research challenges. These can be summarized as follows:

1) A logic-based language must be designed that is free of the sequential execution model and other spurious constructs of PROLOG, without any loss of functionality.

2) Database compilation and optimization techniques must be extended and added to handle the richer functionality of the language.

The desirability of the integration of logic programming and relational database technology has been recognized for some time [Park84]. Attempts to realize such an integration have been so far confined to building interfaces between a PROLOG language-processor and a relational database system [Kun82, Jark84, Bocc86]. This approach suffers from a mismatch between the computational models of these coupled subsystems: PROLOG is oriented towards a tuple at a time model of computation [Zani84], the relational model is oriented towards a set at a time. In the design of LDL we remedy this mismatch by adopting a set at a time model of computation. We realize this model by means of *compilation* i.e., we perform an extensive analysis of the source rules and in this process formulate target-queries in a relational target language. A similar approach to the problem addressed by us is taken in [Ull85, Morr86]. By comparison, PROLOG, not even in its compiled versions [Camp84], makes the distinction between rule-access and data-access and treats these

[1]Turing-complete in this context means that for any given set of base relations $R_1,\ldots,R_n$ and a computable function f there exists an LDL program to compute a derived relation R such that $R=f(R_1,\ldots,R_n)$

as indistinguishable objects. In this paper we concentrate on the functionality of LDL. The compilation techniques are mentioned briefly in sec. 3 and are treated in detail in [Zani85, Banc86].

The remainder of this paper is divided as follows: In section 2 we describe the salient features of LDL and demonstrate these by means of examples. Section 3 gives a brief overview of the compilation technique required for query translation in LDL and we conclude this paper by suggesting some further improvements in section 4.

2. LDL Features.

The approach we have taken in designing LDL is to,

(1) Base the design on a well-understood logic foundation.

(2) Extend this foundation to support language features that are of particular utility to the database application programmer.

Consequently, the design of LDL is based on Horn-clause logic; the logic programming experience has amply demonstrated the utility of this logic as a general purpose programming language. In addition, the semantics of Horn-clause logic is well defined and can be described in operational as well as fixed-point terms [Lloyd84].

At the same time, the design of LDL takes the logic-programming paradigm beyond PROLOG in the following ways:

(1) LDL is based on *pure Horn clause logic*; i.e., the sequential order of execution of rules in a procedure or subgoals within a given rule has been removed.

(2) Sets have been introduced as primitive data objects that can be used directly in the language rather than their simulation through lists as in PROLOG.

(3) A form of negation which is based on set-difference[1] replaces PROLOG's negation by failure.[2]

(4) Schema-definition and update facilities were included.In the following sections we elaborate on each of these features and demonstrate them by means of examples.

2.1 Horn Clause Logic

The sequential order of execution in PROLOG implies that the programmer has the responsibility for the ordering of rules and subgoals within each rule to reflect the problem that he wants to solve. In LDL this responsibility has been moved into the system. The following example is a valid LDL program for the derivation of the `ancestor` relation assuming that `parent(X,Y)` is the base relation.

```
% Example 1: Ancestor Relation.

ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).
ancestor(X,Y) <- parent(X,Y).
```

If a goal specification would be resolved against the rules in this example in PROLOG, then the first clause would be tried before the second one and the computation would not terminate. The order of specification is irrelevant in LDL since *both* clauses are analyzed at compile time, prior to the execution of a query. The result of this analysis is an execution strategy that implements the general fixpoint semantics of these Horn-clauses rather than a strategy which is based on their particular ordering.

Another feature of LDL, which can be supported by Horn-clause logic is the use of *complex terms* in facts and rules. In the following example the emp relation contains information about the first and last names, occupation and education of employees. The use of complex terms enables the grouping of the individual facts about the education of an employee in a flexible way, unlimited by the rigid tuple structure of relational systems.

```
% Example 2: Employee Facts with Complex Terms.

emp(joe,cool,porter,none).
emp(max,fax,guard,high_school(1976)).
emp(joe,doe,vp,college(ms,engl,school(ut,tx),1981)).
emp(fred,red,staff,college(ms,ba,school(mit,ma),1983))
```

Note that the term names used in those facts such as *high-school, college* and *school* serve as placeholders that enable the term-structuring. They do not carry any information and could be omitted. Complex terms should not be confused with (evaluable) function symbols which are not included in the LDL repertoire.

We can formulate rules that can be resolved using complex terms. The following rule specifies the name, school and year of graduation for MBA's who graduated after 1981.

[2]Negation is not in Horn clause logic. This is one instance in which the foundation had to be extended so as to support this feature.

```
% Example 3: New MBA's.

new_mbas(Last,First,School,Year) <-
     emp(Last,First,_,college(ms,ba,School,Year)),
                                          Year > 1981.
```

The query `new_mbas(L,F,S,Y)?` would return the set *{(fred, red, mit, 1983)}* when applied to the fact base of Example 2. Note that the formulation of this problem in a conventional database would force the programmer either to distinguish between employees having a high-school education only and employees having a college education or, to specify null values in his normalized relations and to cope with problems that arise from joining on null values. Neither of these options is very attractive! The rule in Example 3 would then be formulated in a way akin to the concept of generalization as proposed in [Smith 77].

Complex terms can be used in recursive rules. The following is the (by now famous) example of list appending. The list in this example is the complex term and would be represented as *.(x1, .(x2, ...(xn, nil) ...)* where "." is the list concatenation operator.

```
% Example 4: List Append.

append(X,nil,X).
append(X.A,Z,X.B) <- append(A,Z,B).
```

As in Example 1, the order of specification of the clauses is immaterial.

2.2 Sets in LDL

We noted in the introduction that the model of computation employed by LDL is that of a set at a time. Consequently, the response to a query would be to compute *all* of the possible answers that can be deduced from the base relations. In Example 1 above, the response to the query `ancestor(joe,X)?` would be to compute *all* of joe's ancestor and not just an ancestor. In this respect, sets are used implicitly in the computation of LDL queries. LDL provides however an *explicit* form of set manipulation; it enables the user to use sets as data objects in the specification of rules and facts. The advantages of having sets as a primitive in LDL include convenience, expressive power and efficiency. In particular, it allows for the support of nested relations -- a feature which has been advocated by many researchers in the database field e.g., [Banc86a, Ram85, Dada86]. Furthermore, the need for aggregate operations and relational division like operations is evident and supported by most relational systems; these are not expressible by Horn clauses. PROLOG systems have recognized this need too and answered it with an assortment of ad-hoc constructs such as the *bagof* and *setof* primitives, which feature a semantics totally dependent on the sequential execution model of that language. These constructs collect all the terms that satisfy a certain goal into a list. Since a list is a data structure fit for sequential operations only, the potential for parallel execution is therefore lost. We will demonstrate how these limitations can be overcome by means of the three constructs of *set-enumeration, set-generation* and *partition*.

The following example illustrates the use of set enumeration in facts.

```
%Example 5: Set-Oriented Facts.

children_of(joe,mary,{peter,john,lisa}).

employees_of(bill,brown,
          {(red,russell,{working,jogging,bicycling}),
              (mac,fat,{cooking,eating}),
             (graham,greene,{spy_novels}),
               ... } ).
```

In these examples, the first fact states the relationship between parent and their children and the second the relationship between a manager and his employees. In the second example each member of the employee set is represented by a tuple which in itself contains a set of hobbies of the employee. LDL allows thus for the specification of complex terms in facts and rules; these complex terms may include sets. The consequence of admitting sets as data objects is that the unification process which is invoked during goal resolution becomes more complex, i.e., set properties are built in the unification algorithm. For a discussion of the theoretical issues of this problem the reader is referred to [Stick81]. Thus, the query: `children_of(joe,mary,{john,peter,lisa})?` will return "yes" for *any* permutation of the set elements. Other set properties, e.g. associativity, commutativity and idempotence must be recognized as well. We elaborate on these issues in the sequel. Set enumeration can be used in rules as in facts. Example 7 demonstrates the use of set-enumeration in a rule. The query `children_of(Pa,Ma,{Ch})?` when applied to facts of the type in Example 5, would assign a value to the variable Ch and thus return the set of families with *exactly* one child.

Set-generation is the process of generating all of the elements of a set which meet some specification. In standard mathematical notation this would be denoted as $s = \{ x \mid p(x) \}$ where $p(x)$ is a predicate on x. In LDL, set generation is specified in rule form, as follows:

s(< X >) <- p(X).

The following example generates a set of items supplied by a supplier when the base relation is `suppl(Sup#,Item#)`.

```
%Example 6: Set Generation.

item_set(Sup#,<Item#>) <- suppl(Sup#,Item#).
```

The goal: `item_set(S,L)?` will generate an unnormalized relation which collects from the base relation the set of items supplied by each supplier.

In a similar vein, we can generate the *number* of items supplied by each supplier. The following example demonstrates the specification of set-cardinality required to do so.

```
%Example 7: Counting in Sets.

item_count(Sup#,Count) <- item_set(Sup#,S),
                          cardinality(S,Count).

cardinality({},0).
cardinality({X},1).
cardinality(Set,Value) <- partition(Set,Set1,Set2),
                          cardinality(Set1,Value1),
                          cardinality(Set2,Value2),
                          Value = Value1 + Value2.
```

Set cardinality is a second order construct and hence unspecifiable in Horn clause logic. Yet it is obvious that the concept is of great practical utility. Therefore, to include it in the language, we have provided another extension in the form of the *primitive* `partition(S,S1,S2)`. The partition primitive partitions a set S into two disjoint subsets S1 and S2 having at least one element; the exact form of partitioning is transparent to the user. The *partition* primitive enables the specification of the cardinality in a recursive manner and
the operation on each of the partitioned subsets can proceed in parallel [Banc86a]. Other relationships, e.g., aggregates of sets, can be specified in a similar fashion. The recursion is grounded for empty and singleton sets. The notation $\{ X \}$ denotes a singleton set; this is a reference to all sets having *exactly* one element. As such it is an instance of set-enumeration and must be syntactically differentiated from the notion of a set-generator $< X >$ in which we denote a set with an *unspecified* number of elements.

The comparison `Value=Value1+Value2` in Example 7 illustrates the treatment of Arithmetic and comparison predicates in LDL. The "=" sign and other comparison predicates (>, >=, ...) are formally viewed as defined by infinite sets of facts, on two complex arguments, containing all of the argument values that satisfy the relationship. For instance, the set associated with the "<" predicate will not only contain (1<2, 1,<3, ...) but also (1<1+1, ..., 1<3-2, ...) and so on. This view of Arithmetic is cleaner than the arbitrary *is* predicate of PROLOG and is not as complex as a full support of equational logic as e.g. in EQLOG; of course neither is as powerful as the latter since the appearance of functional reduction does not go beyond comparison predicates. Indeed, a goal *eq(2, 1+1)?* expressed against a unit clause *eq(X, X)* will fail in LDL. In practice, the comparison predicates are implemented as built-in operators which are only invoked after the binding of the necessary arguments is complete. The analysis required for the binding-flow is part of the safety check of the compiler (ref. sec. 3).

The following example is another instance of set-enumeration. The relation `3_distinct_parts` derives from the `suppl` relation all suppliers that supply at least 3 distinct parts.

```
%Example 8: Set Enumeration.

3_distinct_parts(Sup#,{X,Y,Z}) <- suppl(Sup#,X),
                                  suppl(Sup#,Y),
                                  suppl(Sup#,Z),
                                  distinct(X,Y,Z).
```

The query, "list all suppliers that supply at least parts a, b, and c" would be formulated as `3_distinct_parts(S,{a,b,c})?` Note that sets may be specified in a non-minimal fashion. In the previous example, the omission of the `distinct` predicate in the right hand side of the rule would cause the possible repetition of elements in the set of the left hand. For unification purposes however these non-minimal sets are equivalent (and thus unifiable) with minimal sets. In the previous example, the assignment $\{X/a,\ Y/a,\ Z/b\}$ would be equivalent to $\{a,\ b\}$.

2.3 Negation.

The form of negation adopted in LDL is based on the computation of set-difference of the relations in the underlying domain. The semantics and power of this form differs from the familiar *negation by failure* [Lloyd84] of PROLOG. The following example demonstrates the use of negation.

```
%Example 9: Use of Negation.

orphan(X)<- person(X), ¬ father(X,Y), ¬ mother(X,Z).
```

Note that this example cannot be handled by NBF since NBF is a test, i.e., ¬ `mother(X,Z)` does not return any bindings for `X` or `Z`. Even if we execute `person(X)` first, the other two goals contain unbound variables and NBF is not defined for goals containing unbound variables. (Lloyd has shown that the soundness of NBF breaks down for non-ground goals).

In our method a given negated subgoal, say $\neg A$ is translated into a relative complement expression, i.e. a *superset* B of A is found and $\neg A$ is replaced by $B-A$. (When such a superset cannot be found the negation is said to be "undefined" or "unsafe"). This notion of supersets, briefly, is based on the following: We view a relation as consisting of a *set* of tuple objects and each tuple being an object. Based on unique identifiers of tuple objects, we can define a set B to be $\geq A$ if all objects in A are included in B. B is then said to be a superset of A. In other words, we impose a partial order $\leq$ on the relations of LDL [Naqvi86].

Thus, in terms of the underlying relations[3] the result to be computed in the above expression,

$$ORPHAN(X) = PERSON(X) - \pi_X\ (FATHER(X,\ Y) \cup MOTHER(X,\ Z))$$

The computation is thus performed using set difference. The use of negation imposes a partial ordering on the computation of the subgoals of a rule: the positive literals are computed prior to their negation; an attempt to compute the negated literal directly by e.g., computing the complement of a set may result in infinite set and generally, to unsafe results [Zani86]. The general case of set-intersection in the presence of negation is typified by the following rule:

$$r(X,\ Y) \leftarrow q(X,\ Y),\ \neg\ p(Y,\ Z).$$

The underlying relational interpretation for this case is,

$$R(X,\ Y) = Q(X,\ Y) - \Pi_{XY}\ (Q(X,\ Y)\ \bowtie_{YY} P(Y,\ Z)).$$

This case covers also the non-intersecting case in the presence of negation.

[3] We will use upper case italics to denote the underlying relations corresponding to the LDL predicates.

Another example is the path between any pair of nodes on a graph where the base relation is the *connect(X, Y)* predicate between a pair of nodes *X,Y*. The *ex_path* relation derives all paths from *X* to *Y excluding* the paths from node *b*.

```
Example 10: Excluding Paths.

path(X,Y)     <- connect(X,Y).
path(X,Z)     <- connect(X,Y),path(Y,Z)
ex_path(X,Y)  <- path(X,Y), ¬ path(b,Y).
```

This example can be computed using the same interpretation of the previous case. To illustrate this process, consider the following specific example of an *ex_path* computation. Figure 1 depicts a simple graph:3

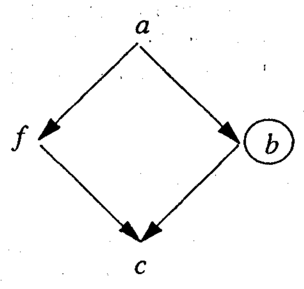

Figure 1: A Simple Graph. The Excluded Node is *b*.

For this graph, *connect(X,Y)* = {*(a,b), (a,f), (b,c), (f,c)*}. The path relation is the transitive closure over *connect(X,Y)* and is: *path(X,Y)* = {*(a,b), (a,f), (a,c), (b,c), (f,c)*}. Assuming that node *b* is the excluded node, then, *path(b, Y)* = {*(b, c)*}.The join of *path(X, Y)* and *path(b, Y)* yields: {*(a, c, b, c), (f, c, b, c), (b, c, b, c)*}; and the projection over *X* and *Y* of the joined relation is {*(a, c), (f, c), (b,c)*}. Finally, the set difference between the projection and *path(X, Y)* is {*(a, b), (a,f)*}.

Note that a simple set difference between *path(X, Y)* and *path(b, Y)* yields {*(a,b), (a, f), (a, c), (f, c)*} which is wrong since the paths *(a, c), (a, f)* and *(f, c)* have been left in the result.

2.4 Updates.

A complete update capability in a logic based data language supports the updating (adding, deleting, changing) of:

(1) The database schema.

(2) Base relations

(3) Derived relations.

The present version of LDL provides only a minimal facility for updates on schemata and base relations. Database schemata can be updated by the following primitives:

create(RelationName, NrArguments).

destroy(RelationName).

These primitives contain the integrity checks required for the maintenance of unique relation names. Relations can be loaded/unloaded from/to files by the primitives:

load(RelationName, File).

unload(RelationName, File).

Beyond the atomic updates to relations (insert, delete and change a tuple in a base-relation) the problem is still open and subject to current research. The issues addressed are:

(1) The specification of *composite updates*. How can the (partial) results of an update be committed so as to be visible to the subsequent update(s) in a composite update? The commitment should be data-driven and consistent with the logic-programming paradigm.

(2) The parametrization of composite updates. How can a composite update be specified so as to avoid the re-specification each time a new set of relations is involved? This is a software-engineering issue as opposed to the previous issue which has to do with semantics.

3. LDL Compilation Techniques

We noted in the introduction that we have adopted a compilation approach to achieve the twofold objective of set-at-a-time model of computation support and the high performance execution of the queries. In this section we present a brief overview of the compilation techniques employed. The reader is referred to the referenced work for the details of each of these techniques. The compilation process proceeds in two phases:

(1) The compilation of the rule-set.

(2) The compilation and optimization for queries.

The rule-set is transformed into a *predicate connection graph*. This structure stores the relationships between terms and the clause-heads that can (potentially) be unified with them. In addition it serves to maintain the entry points for individual queries. For a description of this structure and its maintenance under a changing rule set see [Kell86]. The factors that determine the query compilation process are the *rule complexity* and *term complexity* of the rule(s) and their arguments which are invoked by the query. The techniques required for the compilation of *recursive rules* by means of naive evaluation and magic sets is described in [Banc85, Banc86]. The problems addressed by these techniques are the propagation of select operations into the recursive structure and the early elimination of redundant tuples which do not contribute to the result-set. The techniques required for the unification of *complex terms* i.e., terms representing unnormalized relations, and their translation into an extended relational algebra are described in [Zani85]. Another compile-time analysis which is required is for *safety* purposes. This analysis, which is of special importance in the presence of arithmetic predicates, ensures that the generated result-sets for a query are finite. The following is an example of an unsafe rule:

p(X, Y) <- X = Y.

The query p(3, Y)? is safe whereas the query p(X, Y)? is unsafe in the sense that it may generate an infinite set of answer tuples; the method for detecting unsafe queries and compiling safe ones is described in [Zani86].

4. Conclusion

In this paper we have outlined the motives for the design and have shown some of the main features of LDL. The design of this version of the language is complete and we have encoded two prototype applications in LDL for verification purposes. These applications pertain respectively to navigation on a map and an order-entry system [Keller86]. In the map case the stored data contains adjacency relations between streets and other pertinent relationships ("Yellow Pages"). Various heuristics are formulated in LDL for the improvement of the search process over this map. The order entry application contains a rule-set for the routing / inventory maintenance of stock. In addition, a set of standard transactions e.g., "new order" are formulated. Experience so far demonstrates that the design is indeed adequate for the intended, knowledge-intensive applications. At the same time the accumulated experience points to a number of improvements that will be included in the next version of the language. These improvements are:

(1) Named attributes: experience shows that the positional notation adopted in LDL is occasionally awkward. Particularly when the predicates have a large number of arguments and only a few need to be specified. We will include therefore a capability to name the arguments.

(2) A type system: There is a strong need for a built-in inheritance capability in the language. Although inheritance can be explicitly encoded in LDL it would be more efficient and natural to subsume it in the language. The work on LOGIN [Ait85] is of particular importance in this respect.

(3) KBS primitives: LDL contains a number of primitives such as *partition* and the update primitives. More of these are required, in particular for such tasks as AI knowledge representation: frames and others.

(4) An expanded update capability: we noted that our update facility is incomplete. We will expand it to include updates on derived relations as well as a more elaborate schema definition facility.

To conclude, LDL has been validated as a powerful and flexible tool for the specification of knowledge-intensive applications. The language evolution process in which more and more of the functionality will be removed from the users' responsibility to the system is ongoing and will reflect itself in future designs.

Acknowledgement

This paper has greatly benefited from the critical reading and suggested improvements by Shamim Naqvi. The authors wish to thank him for his efforts in this respect.

References

[Ait85] Ait-Kaci, H., R. Nasr "LOGIN: A Logic Programming Language With Built-In Inheritance". *MCC Technical Report* MCC-AI-068-85, 1985 (To appear in the *Journal of Logic Programming).*

[Banc85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", *MCC Technical Report,* DB-004-85, 1985.

[Banc86] Bancilhon, F., D. Maier and J. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc ACM SIGACT / SIGMOD symposium on Principles of Database Systems,* Washington, 1986.

[Banc86a] Bancilhon, F. and S. Khoshafian, "A Calculus for Complex Objects" *Proc ACM SIGACT / SIGMOD Symposium on Principles of Database Systems,* Cambridge, MA., 1986.

[Bocc86] Bocca, J., "On the Evaluation Strategy of EDUCE", *Proc. SIGMOD Conference on Management of Data,* Washington, D.C., 1986.

[Camp84] Campbell, J. A., *Implementations of PROLOG,* Ellis H. Horwood Publ., 1984.

[Clock84] Clockskin, W. F. and C. S. Mellish, *Programming in Prolog,* Springer Verlag Publ., Heidelberg, 1984.

[Dada86] Dadam, P. et al., "A DBMS Prototype to Support Extended NF 2 Relations: An Integrated View on Flat Tables and Hierarchies" *Proc. ACM SIGMOD Conf. on Management of Data* , Washington, 1986.

[Gall78] Gallaire, H. and J. Minker, *Logic and Databases,* Plenum Publ., 1978.

[Gall84] Gallaire, H., J. Minker and J. Nicolas, *Advances in Database Theory,* Vol. 1 & 2, Plenum Publ., 1984.

[Gall84a] Gallaire, H., J. Minker and J. Nicolas, "Logic and Databases: A Deductive approach", *Computing Surveys,* Vol 16, #2, June 1984.

[IBM81] "SQL/data system: Concepts and facilities", *GH24-5013-0,* File No. S370-50, IBM, 1981.

[Jark84] Jarke M., J. Clifford and Y. Vassilou, "An Optimizing Prolog Front-end to a Relational Query", *Proc. ACM SIGMOD Conference on Management of Data,* Boston, MA., 1984.

[Kell86] Kellogg, C., A. O'Hare and L. Travis, "Optimizing the Rule-Data Interface in a KMS" Submitted for Publication.

[Keller86] Keller, T., et al. "ADBS Workloads, Rev. 0.5" *MCC Technical Report,* DB-006-86, 1986.

[Kowa79] Kowalski, R. A., "Algorithm = Logic + Control", *Comm. ACM,* August 1979.

[Kun82] Kunifuji, S. and H. Yokota, "Prolog and Relational Databases for 5th Generation Computer Systems", *Proc. Workshop on Logical Bases for Data Bases,* Toulouse, France 1982.

[Morr86] Morris, K. Ullman J.D. and A. Van Gelder, "Design Overview of The NAIL! System", Unpublished Manuscript, Stanford University, 1986.

[Lloy84] Lloyd, J. W., *Foundations of Logic Programming,* Springer Verlag Publ., Heidelberg, 1984.

[Naish85] Naish, L., "All Solutions Predicates in Prolog", *Proc. 1985 Symposium on Logic Programming,* Boston, MA., 1985.

[Naqvi86] Naqvi, S., Report in preparation.

[Park84] Parker, D.S., et al. "Logic Programming and Databases", *Proc. First International Conference on Expert Database Systems,* Kiawah Island, S.C., Oct. 1984.

[Quin85] "Quintus Prolog Reference Manual", Quintus Computer Systems, 234 Yale Street, Palo Alto, CA., 94306.

[Ram85] Ramakrishnan, R. and A. Silbershatz "The MR Diagram -- A Model for Conceptual Database Design", *Proc 11th Conf. Very Large Data Bases,* Stockholm, 1985.

[Smith77] Smith, J.M. and D.C.P. Smith "Database Abstractions: Aggregation and Generalization" *ACM Trans database Systems* 2, 2, pp. 105-133, 1977.

[Stick81] Stickel, M. E., "A Unification Algorithm for Associative-Commutative Functions", *JACM,* Vol. 28, #3, July 1981.

[Ull85] Ullman, J.D., "Implementations of Logical Query Languages for Databases"*ACM Trans on Database Systems* Vol 10 #3, Sept 1985.

[Zani85] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects", *Proc. 11th Int. Conf. very Large Data Bases,* Stockholm, 1985, pp. 458-459.

[Zani84] Zaniolo, C., "Prolog: A Database Query Language for All Seasons" *Proc. First International Conference on Expert Database Systems,* Kiawah Island, S. C., October 1984.

[Zani86] Zaniolo, C. "Safety and Compilation of Non-Recursive Horn-Clauses" *Proc. First International Conference on Expert Database Systems* Charleston, S.C., 1986.