DESIGN AND IMPLEMENTATION OF A LOGIC BASED LANGUAGE FOR DATA INTENSIVE APPLICATIONS

Carlo Zaniolo

Microelectronics and Computer Technology Corporation
Austin, Texas, USA

Abstract

Ongoing research on the Logic Data Language (LDL) pursues a new application focus and a new implementation technology for Logic Programming. In LDL, the functionality of Prolog is enhanced with full database capabilities, such as, schema facilities for extensional information, transaction management and recovery. The implementation technology of LDL is, however, very different from that of Prolog, since it is based on mechanisms suitable for secondary storage, such as matching and least fixpoint computations, rather than on SLD-resolution and unification. This paper describes the main research challenges tackled and the solution approaches taken in designing and building two experimental LDL systems. Thus, the following topics are discussed: (a) the design of non-Horn constructs (such as negation, sets and updates) and the definition of a formal semantics for them, (b) the compilation methods that map rules and queries with mode declarations into efficient execution plans based on matching and fixpoint operators, (c) the optimization and safety techniques used to relieve the user from the responsibility of specifying an execution order for clauses and goals in the program and (d) the new style of deductive programming that emerges from writing applications in LDL.

1. Background

The research on the Logic Data Language (LDL) is motivated by the conviction that data intensive applications represent a natural application domain with great growth potential for Logic Programming. This motivation is not unique to the LDL work, since it can be seen as the rationale behind several systems designed for coupling Prolog with relational database systems [Boc, CeGW, KuYo, Li, JaCV].

LDL is, however, unique in its technical approach that is based on the tenet that a truly effective logic based language for data intensive applications cannot be built by coupling existing Prolog systems with

database systems, rather, new language concepts and a new implementation technology are needed. This tenet has led to the design of a novel language based on Horn clause logic —with its distinctive style of application development— and to a novel repertoire of techniques for efficient implementation of logic based languages. This paper provides a reasoned history of the evolution of LDL, an overview of the research challenges encountered and of the solutions developed in the design and implementations of two experimental LDL systems.

The basic LDL idea, conceived in 1984, envisions a system that combines the expressive power of Prolog with the functionality and facilities of Data Base Management Systems (DBMSs) —i.e., support for transactions, recovery, and schema based integrity and efficient management of secondary storage. Therefore, the LDL system offers a user all the benefits of a database language [BaBu] including the elimination of the "impedance mismatch" between the programming language and the database query language [CoMa], which is besetting the development of data intensive applications. Moreover, LDL represents an important way-station toward future Knowledge Management Systems which will have to combine efficient inference mechanisms from Logic with efficient and secure management of large information banks from Database Systems.

It soon became clear that arduous research challenges stood in the way of realizing the LDL concept. The first issue that came into focus was that of users' responsibility of execution control. In the '70s and early '80s, the database field had witnessed a dramatic evolution from navigational systems into relational ones. In navigational systems, such as Codasyl compliant DBMSs, the programmer must explicitly navigate through the maze of database records, paying careful attention to the sequential order in which these records are visited —the key to efficiency. In relational DBMSs, instead, the user is only responsible for the formulation of a correct query (using logic-based languages of limited expressive power, such as SQL or QUEL [Ull]). A special system module, called the query optimizer, then compiles each query into an efficient execution plan. By contrast, in Prolog, the programmer must order carefully rules and goals to ensure efficient execution and termination. This basic mismatch, from which all systems coupling Prolog with relational DBMSs suffer, also challenged LDL's quest for a harmonious integration, leaving two alternative paths open [Zan1]. One consisted in adding navigational database facilities to a Prolog-like language; the other in rejecting the navigational (procedural) semantics of Prolog, in favor of a purely declarative one, whereby the order of goals and rules in a program becomes immaterial. In the fall of 1984, the critical decision was taken to pursue the second solution, with the expectation that it will deliver better usability and potential for massive parallelism and lead to more exciting research problems and technology break-throughs. As it is described next, this early decision

had profound repercussions on both the design of the language and its implementation.

A Prolog programmer must be keenly aware of its sequential execution model (SLD-resolution where the leftmost goal and the first rule is selected [vEKo,Llo]), not only because the termination and performance of the program will depend on it, but also because the very semantics of the many non-Horn constructs —primarily cuts, and updates, but also negation and "set-of" predicates— are based on such execution model. These non-Horn constructs were introduced in Prolog to obtain the expressive power needed for application development. Having decided to divorce execution from the order of rules and goals in the program, the first technical challenge facing LDL research was to provide a clean design and a formal declarative semantics for the non-Horn constructs that were needed in the language for reasons of expressive power. The result is a language that is different from Prolog in terms of the constructs and programming style it entails. Section 2 of this paper focuses on the language design issues. Section 4 describes the optimization techniques used to derive safe and efficient execution schemes for LDL programs.

The major design choices regarding the LDL implementation approach, crystalized in the Spring of 1985. These choices reflect the emphasis placed on efficient support for data intensive applications. The need to update a large fact base frequently and efficiently dictated that a sharp separation be drawn between the data and the program. Thus, in LDL, the fact base is described through a database schema definition facility, and can be changed without requiring program interpretation or recompilation. Furthermore, it was decided that, to obtain maximum performance on secondary storage resident data, the execution model should avoid SLD-resolution and unification. This led to the definition of a simpler execution model that is based upon the operations of matching and the computation of least fixpoints. An immediate benefit of this approach is that matching operators on sets of facts can be implemented using simple extensions to the Relational Algebra [Zan2, Zan3], which is a target language of proven effectiveness in accessing databases on secondary storage [Ull].

Having chosen a simpler target language, the LDL designers were faced with the challenge of designing a more sophisticated compiler to support the full functionality of the source language. The solution approach chosen is built on two pillars:

(i) the use of a global analysis to infer the bindings induced by a specific query in rules and goals, and

(ii) the compilation methods which rewrite recursive programs, that, as such, are not efficient or safe to implement by fixpoint computations, into equivalent programs that are.

Sections 3 and 5 of this paper, respectively, describe the compilation techniques used and the overall architectures of two experimental LDL systems (one for a highly parallel database machine, the other for a Unix workstation). The final section describes the new style of programming with Logic that emerges from the declarative nature of LDL and our initial experience in developing applications in the language.

2. Language Design: the Importance of a Clean Semantics.

The LDL language design reflects a desire for clarity in syntactic constructs and formality in the semantic definition. For instance, in current Prolog systems, it is not immediately clear whether variables in negated literals are existentially or universally quantified, and the answer may change with the order of goals. LDL's syntax removes any room for ambiguity by simply assuming that all variables are universally quantified.

Sets are supported in LDL by a first class repertoire of declarative constructs. A set can be described by listing its elements within braces —*set enumeration* construct. For instance, the following two facts state that Mark likes Mary, Janet and Erica, while Tom likes Mary and Janet.

likes (mark, {mary, janet , erica}).
likes (tom, {mary, janet}).

Obviously, neither boy shows any preference among the girls he likes, inasmuch as these are not lists, but true sets where the order of elements is immaterial.

The *set grouping* construct is available to collect into a set all elements satisfying a certain property. For instance, to define a relation where for each girl there is the set of boys liking her, we can write:

is-liked(<Boy>, Girl) ←
likes(Boy, Girl_Set), member(Girl, Girl_Set).

Thus the query,

?is-liked(Boys, mary)

will return Boys = {mark, tom}. Thus, set grouping is denoted by a pair of pointed parentheses, <>, in the head of the rule —no such construct is allowed in the body of a rule.

The meaning of programs with set constructs and negation is defined by a formal semantics that extends the model theoretical and fixpoint-based semantics of Horn Clauses [vEKo, Llo]. This extension is based on the notion of stratification, [ApBW, Naq, Prz] which is best explained by a simple example. The following program (this word is here used as a synonym of a set of rules) defines, for each person X, all the persons that are not his/her ancestors:

```
unanc(X, Y)   ← person(X), person(Y), not(anc(X, Y)).
anc(X, Z)     ← anc(X, Y), parent(Y, Z).
anc(X, X)     ← person(X).
```

Figure 1. *Ancestors and Non-Ancestors*

The dependency graph for the program of Figure 1 is given in Figure 2.

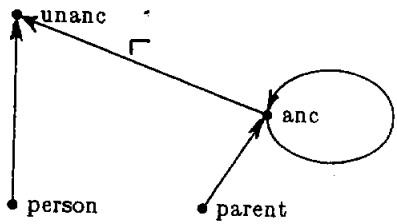

Figure 2. *Dependency graph for the program of Figure 1.*

Thus a dependency graph for a program P has as nodes the predicate symbols of P, and, for each goal in P, it contains an arc from the goal predicate symbol to the head predicate symbols. If the goal is negated, then the edge is labelled. The maximal strong components of the dependency graph identify the recursive predicate symbols in the program; we will refer to them as *recursive cliques*. For instance, the recursive clique of Figure 2, contains only one node, thus it defines a simple recursive predicate; recursive cliques with two or more nodes define mutually recursive predicates. A rule defining a recursive predicate is called a *recursive rule* if its body contains some recursive predicate from the same recursive clique as the head predicate, and it is called an *exit rule* otherwise. The notion of recursive clique is important both from a semantic viewpoint and an implementation viewpoint.

A program in which there are no labelled arcs in any recursive clique is called *stratified*. Stratified programs can be partitioned into layers such that each (negated) predicate in the body falls in a layer that is (strictly) lower than that of the predicate in the head. This layering induces a unique standard choice among alternative minimal models, known as *perfect model* [Prz]. Perfect models capture the intuitive semantics of programs, and can be computed by a succession of least fixpoint operations. For instance, the program of Figure 1 is stratified since its only cycle contains no labelled arcs. Its perfect model can be derived by computing first the lower layer containing anc, and then the higher one containing unanc —bottom-up computation. Programs which are not stratified often neither have a formal semantics nor a clear intuitive meaning. Thus, in LDL, the only admissible programs are the stratified ones.

The definition of the formal semantics for sets is given in [Bet1]. The notion of stratification is also important in this context; it is

employed to disallow the use of the set grouping operator in the definition of recursive cliques, since this could lead to various paradoxes [Bet1, ShNa].

A particularly challenging problem in the design of LDL was the definition of updates. These are designed to modify the fact base, not the rule base, and offer full support for the notions of transaction and recovery. A first solution to the update problem, proposed by French researchers [deSi] use the head of a rule to describe the update action —as if it were a production rule. A second approach, favored by Australian researchers, limits the use of updates to the main query conjunct [Reta, NaTR]. Both approaches were discarded for LDL, since they do not support the passing of run time values to generic update procedures. To overcome this limitation, LDL allows updates in the bodies of rules. For instance a rule †:

```
happy(Dept, Raise, Name) ←
        emp(Name, Dept, Sal), Newsal = Sal+Raise,
        -emp(Name, Dept, _), +emp(Name, Dept, Newsal).
```

combined with

```
?happy(software, 1000, Name).
```

will give a $1,000 raise to all employees in the software department and return the names of those happy employees. The query above is regarded as an indivisible transaction; thus, if any of the specified updates cannot be completed, then the whole transaction is aborted and the database is returned to the original state. This situation will, for instance, occur when some integrity constraint is violated, or another run time error occurs in the computation of the rule, (e.g., if a non-numeric Sal value stored for some employee in the Software department makes it impossible to perform the addition "Sal + Raise").

Update actions are formally defined using dynamic logic [Har], where each update goal is modeled by a [before:after] pair [NaKr]. For goals other than updated goals, the before and after states are the same, and the old semantics is recovered [NaKr]. The meaning of rules with several updates goals, however, can depend on the order in which these appear in the rule. For the rule above, for instance, the goal -emp(Name, Dept, _) must be executed before the +emp(Name, Dept, Newsal) goal (switching the order of the two will result in the firing of all employees from the software department).

† An expression, such as "Newsal = Sal+Raise", is used in LDL both to instantiate Newsal to the value resulting from the evaluation of the right hand side expression and to verify the equality of such value with a Newsal value instantiated by other goals.

Therefore, some interesting lessons have emerged from the LDL search for a formal declarative semantics for various non-Horn logic constructs. While the model theoretic semantics of Horn clauses has been successfully generalized, the resulting formal semantics takes a procedural flavor when updates are present, since it models a sequence of computation steps that is based on the explicit order of rule goals. This can be contrasted with the implicit ordering of computation imposed by the notion of stratification used for negation and set-grouping. These two alternatives, i.e., the implicit versus the explicit sequencing of computation steps, can also be found in more recent proposals aiming at extending the power of logic programming by allowing a more liberal use of negation in recursive predicates. Thus, an explicit control through a rule algebra is proposed in [ImNa] as the means to overcome the ambiguities caused by negation in unstratified programs. Implicit control via index counters, is instead proposed in [SaZ5] as the means to achieve computable perfect models for a wider class of recursive programs with negation. While a lot of attention has been given to the issue of allowing more liberal use of negation in recursion [Prz, KoPa], the problem of relaxing stratification for set grouping seems less pressing, perhaps because some key theoretical issues on the expressive power of these programs are not yet resolved [ShNa].

Finally, LDL offers an if-then-else construct of clean declarative semantics, for the clear expression and efficient implementation of mutually disjunctive rules [Okee]. In addition, it offers a non-procedural *choice* predicate, for situations where any answer will do [KrNa]. This useful declarative device, which eliminates any residual need for having a cut construct in LDL, is discussed in Section 5. The choice predicate can also be used to obtain a single answer response, rather than the all-answer solution that represents the default response for LDL queries.

3. Compilation: a Discourse on Methods

A first task of the LDL compiler is to parse the rule base and to generate a Predicate Connection Graph (PCG) representing these rules [KeOT]. The compilation proper begins when a query form is given, i.e., a query with mode declarations specifying the arguments that will be given (ground) at actual query time. Then the constant migration step for non-recursive predicates is performed. For instance, consider the query form

?grandma($X,Y).

(where $X denotes that a value is to be supplied at actual query time) and the following set of rules:

implementation. A considerable amount of research has been devoted to this key problem [HeNa, BMSU, GaDe, KiLo, Vie, SaZ1]; the reader is referred to [BaRa] for an overview of these techniques. The LDL compiler uses the *magic set method* [BMSU, SaZ2] and the *generalized counting method* [SaZ3], which are expressible by rule rewriting scripts and lead to efficient implementations using fixpoint computations. In a nutshell, these methods take a recursive clique that, for the given query, cannot be supported well by means of a fixpoint computation and recast it into a pair of connected recursive cliques, each amenable to efficient fixpoint implementation.

This transformation can be illustrated by the example where people of the same generation as Marc are sought. One alternative way to find these people consists in

(i) deriving the ancestors of Marc and counting the levels as we go up (Marc being a zero level ancestor of himself).

(ii) once an ancestor of Marc, say X, is found, then the descendants of X are computed, while levels are counted down. Descendants for which the level counter is zero are of the same generation as Marc.

We can express the previous computations as follows (J+1 and J-1 denote the respective successor and predecessor of the integer J):

```
sg.up(0, marc).
sg.up(J+1, XP)   ← parent(X, XP), sg.up(J, X).
sg.down(J, X)    ← sg.up(J, X).
sg.down(J-1,X)   ← sg.down(J, YP), parent(Y, YP).
?sg.down(0,X).
```

Thus the initial recursive clique has been reformulated into a pair of recursive cliques connected via the index J. Each recursive clique can now be implemented efficiently and safely using a fixpoint computation (indeed each is basically a transitive closure operation).

The equivalence preserving transformation that we have just introduced using the intuitive semantics of ancestry, can be performed with full generality on a purely syntactic basis. Indeed, observe that in a succession of recursive Prolog calls generated by the goal sg(marc, X), X and XP are bound and Y, YP are not. Thus, the recursive sg.down rule is basically constructed by dropping the bound arguments and retaining the others, while a new argument is added to perform the count-down. The recursive rule for up.sg is instead built by retaining the bound arguments and then exchanging the recursive predicate in the head with that in the tail of the rule (indeed, we want to simulate a top-down computation by a bottom-up one), and then adding the count-up indexes. Also observe that the original exit rule is used to glue together the up and down computations. Finally, the bound part of the query goal becomes the new exit rule

for up.sg, while the unbound part becomes the new query goal. The generalized and formal expression of these rule rewriting techniques, known as the generalized counting method are given in [SaZ3].

The counting method is very efficient for acyclic databases, but will loop forever, as Prolog does, for cyclic databases, e.g., for the same-generation example above, if parent has cycles. The magic set method solves this problem by first computing all the ancestors of Marc and then using these to restrict the computation of the original rules. Thus, for the previous example, the magic set method will produce:

```
m.sg(marc).
m.sg(YP)   ← parent(X, XP), m.sg(X).
sg'(X,X)   ← m.sg(X).
sg'(X,Y)   ← m.sg(X), parent(X, XP), sg'(XP, YP), parent(Y, YP).
?sg'(marc, X).
```

Observe that the computation has again been broken down into a pair of recursive cliques. The computation of Marc's ancestors expressed by m.sg will terminate even if the graph corresponding to the relation parent is cyclic. Moreover, the addition of the new goal to the original rules make them safe and more efficient to execute; e.g., the X values in exit rule for sg' now range over the ancestors of Marc, and not over the whole Herbrand universe. In passing, we also mention the *magic counting method* [SaZ4] that combines the advantages of two methods just discussed. Since the computation of the first fixpoint of the magic set method is very similar to that of the counting method, there is no need to commit to one method at compile time. Rather, it is possible to switch to the magic set method at run time when the presence of a cycle is detected, or simply suspected [SaZ4].

In many situations of practical import, two fixpoint computations are not necessary and it is possible to rewrite the rules in such a way that one will do. For instance, the query

```
?anc(marc, Z).
```

can be supported by specializing the anc rules of Figure 1 into,

```
anc(marc, Z)     ← anc(marc, Y), parent(Y, Z).
anc(marc, marc)  ← person(marc).
```

and then dropping the constant argument to yield:

```
anc'( Z)     ← anc'( Y), parent(Y, Z).
anc'(marc)   ← person(marc).
```

A fixpoint computation implements this transitive closure operation very efficiently, since the original query condition is now applied directly to the datum parent relation and not the derived anc relation (i.e., selection

has been pushed inside recursion). More complex rewriting is required for the query,

?anc(X, brian).

where, before the specialization approach can be applied, the recursive rule must be rewritten in its right-linear form, as follows:

anc"(X, Z) ← parent(X,Y), anc"(X, Z).

The LDL compiler is capable of recognizing this and many other simple but important cases where some transformation of the original program will reduce it to one implementable efficiently via a single fixpoint computation. In general however, the problem of recognizing when such a transformation exists is known to be undecidable [Bet2].

While no function symbols were present in the previous examples, all the compilation techniques just described apply when these are present. Take for instance, a fast list reverse computation,

?revzap([a,b], [], X).
revzap([X| L], L2, L3) ← revzap(L, [X| L2], L3).
revzap([], L, L).

This will be rewritten and implemented as follows:

revzap'([a,b], []).
revzap'(L, [X| L2]) ← revzap'([X| L], L2).
?revzap'([], X).

In summary, the recursive compilation methods just discussed reduce the problem of supporting recursive cliques to that of computing least fixpoints efficiently. However, the standard fixpoint algorithm is, as such, inefficient since it does redundant work. For instance in the ancestor example, the j-th step of the fixpoint iteration applies the immediate consequence operator, T_P [Llo], to all ancestors computed so far, i.e., to ancestors of every level between zero and j, although only the j-th level ancestors are needed to compute those at level $j+1$. An improvement of the fixpoint algorithm known as *semi-naive* fixpoint can be used to solve this problem [Ban, BaR]. This improvement is based on symbolic finite differentiation techniques. While some formulations propose rather cumbersome differentiation on the relational algebra equivalent of T_P [Ull], the differential fixpoint method described in [SaZ5] operates directly as a rule rewriting method.

Another area of considerable innovation in the LDL compiler is the support for set terms. Set terms are treated as complex terms having the commutativity and idempotence property. These properties are supported via compile time rule transformation techniques, that use sorting and various optimization techniques to eliminate blind run time searches for

commutative and idempotent matches [ShTZ].

4. The Optimizer: Exercising Self-Control

The issue of control has received considerable attention in the past. One research direction has focused on giving the programmer more options and explicit control. In terms of automatic control, most of the previous research concentrates on exercising run time control. For instance, a common approach to safety consists in freezing the execution of unsafe goals until enough arguments are instantiated [AiNs, Col,Nai]. Reasoning at the metalevel is suggested in [SmGe] as the means for deciding which goal to expand next. LDL's integrated approach to safety and optimization is instead a compile-time approach that builds upon the successful experience gained with relational database systems. This experience led to the characterization of the optimizer in terms of:

(i) An *execution model,* whereby the execution of a query can be described by an abstract execution graph, such as an operator tree. The set of (logically) equivalent execution graphs for a query defines the search space for the optimization,

(ii) The *cost functions,* whereby a cost estimate is associated with each point in the search space, and

(iii) The *search strategy,* to determine the minimum cost execution in the given space.

The predicate connection graph for the query after constant migration supplies the basis for the execution model. This graph is then reduced to an AND/OR tree by (1) contracting the recursive cliques into a single node and (2) transforming the resulting DAG into a tree. The resulting AND/OR tree can then be viewed as a relational algebra expression since the disjuncts map into union operations, while the conjuncts map into join operations (except for negated goals that define set difference operators). The order in which joins are performed is of the greatest importance, while the order in which unions are computed is considered of little importance. Thus the search space for the optimizer includes all possible permutations of goal orders in the rules. Furthermore, there are basically two ways to perform a join. One is to materialize all operands first and then to perform the join. The other is to materialize only the left operand, or possibly its next tuple, and then to find the matching values in the right operand —thus realizing what is known as a Sideways Information Passing (SIP) strategy [BeRa]. Furthermore, if the right operand is defined as the union or join of lower operands, we need not materialize it; rather the the SIP can be applied to these lower operands. Repeated applications of this strategy lead to an execution strategy similar to the nested loop joins of Prolog. Thus, the optimizer will evaluate, for each join, whether to use a materialized or a SIP strategy. The optimizer will also have to

select a strategy for the processing of each recursive clique. The order of joins and their execution strategy (SIP or materialized) determine the bound arguments in the recursive goals. Thus, the optimizer will expand the corresponding recursive clique, determine the safety of applicable compilation methods, and then choose from the safe methods on the basis of their cost.

As part of the challenge of building an optimizer for LDL, cost functions had to be provided for heuristically effective cost estimates for arbitrary AND/OR trees and recursive cliques. Of particular interest are the issues of determining the termination of a method used for a recursive clique and of estimating the cost of the resulting fixpoint computations. The comprehensive heuristics used by LDL in dealing with the undecidable problem of safety are described in [KrRS]. Executions that are unsafe (i.e., do not terminate) are assigned an extremely high cost, to ensure that they will be discarded by the search strategy.

The simplifications used in deriving simple formulae and criteria for cost estimates and safety are frequently crude. However, similar approximate estimates based on database statistics have been effective in relational databases, their coarseness not withstanding. Indeed, the effectiveness of an optimizer is not to be measured by its ability to find the actual optimum, but rather by its ability to consistently devise execution strategies of acceptable performance. This consideration is also relevant in evaluating the search algorithms discussed next.

Relational systems optimizers are based on an exhaustive search, improved with dynamic programming techniques that reduce the search from factorial to exponential [Seta]. This approach becomes prohibitively expensive for large LDL programs. Therefore, two alternative methods are currently under evaluation. One is a quadratic time algorithm that delivers optimal ordering for acyclic queries and semi-linear cost functions [KrBZ], and it is also heuristically effective for the general case [Vil]. Encouraging results were also obtained with a stochastic algorithm based on Simulated Annealing [IoWo, KrZa].

5. Experimental Systems: David and Goliath

The biblical simile underscores the difference in size of the two experimental implementations of LDL currently being undertaken. The large system is based on a massively parallel multiprocessor system that manages large databases residing in secondary store. The main responsibility for the LDL compiler/optimizer is to map LDL code into a relational algebra based language called FAD [DaKV]. The responsibility for efficient parallel execution is up to the FAD compiler. The LDL system consists of the following modules:

1) the user interface,
2) the schema manager,
3) the rule manager,
4) the query form manager,
5) the optimizer, and
6) the query manager.

The rule manager and the query form manager perform the actual compilation of LDL into FAD. The query manager handles the precompiled object modules and selects the proper module for execution, by matching the actual query with the precompiled query forms.

The small system is a complete implementation of LDL for a Unix workstation. This experiment is motivated by the desire of having (i) a more portable demonstration vehicle for LDL and (ii) a test-bed to quantify the performance gains resulting from the sophisticated compilation techniques developed for the LDL compiler. Thus, while the compiler/optimizer for the new machine is similar to that of the larger system (the main changes pertaining to the code generator) the execution model and target language for the new machine are very different. The separation between data and program is still present, as reflected by the two main components of the run time system being an Abstract Machine (AM) and a Fact Manager (FM). However, the set-at-a-time execution of relational algebra is replaced by a tuple-at-a-time execution, since the data is assumed to be accessible in main memory. The extensive compile time analysis entails a more static memory management and improved performance ‡.

6. Deductive Programming in LDL

While LDL's theory builds upon the formal work on deductive databases [GMN, Rei], its pratical focus on application development and efficient implementation aligns it with the main stream of the Logic Programming paradigm. Therefore, it is of interest to compare LDL with Prolog from a programmer's viewpoint. The experience of programming in LDL is still limited, but a reasonable understanding is emerging on the programming style it entails, as illustrated by the recent textbook on LDL [NaTs]. For simple examples, programming in LDL is very similar to programming in Prolog —so much so that the first chapters of any current logic programming textbook can be used as a primer on LDL. But the gap becomes wider as the sophistication of the application grows. The first important difference comes with recursion. A procedure, to generate

‡ The author jokingly refers to the resulting abstract machine, which is much simpler than that in [War], as "a FORTRAN Technology Theorem Prover".

all integers between zero and a given integer K, can be expressed as follows in LDL:

```
int(K, 0)   ← K >= 0.
int(K, J)   ← int(K, I), I<K, J= I+1 .
```

This represents a very natural rendering of Peano's inductive definition of integers, augmented with a condition on K in the second rule to ensure termination, and one in the first rule to ensure that no answer is returned for a negative K. The fixpoint implementation of this recursive procedure by the LDL compiler is moreover very efficient since it only uses a basic iteration. A second formulation is also possible in LDL, as follows:

```
int(K,J)    ← K>0, K1 = K-1, int(K1, J).
int(K,K)    ← K >= 0.
```

This is a less clear and intuitive definition, but it is the only one that can be handled by Prolog (the equal signs would also be replaced by "is").

A second interesting example is supplied by the computation of Fibonacci numbers.

```
fib(0,1).
fib(1,1).
fib(I+1, N1+N2) ← fib(I, N1), fib(I-1, N2).
```

This program also illustrates the little sugaring of the syntax that, in LDL as in other systems [Col], allows arithmetic expressions in predicate arguments. This program, presented with a query ?fib(I,N) will compute all Fibonacci numbers. A more practical version of this is the following one that computes all Fibonacci numbers up to an certain given integer K:

```
fibn(K,0,1) ← K>=0.
fibn(K,1,1) ← K>=1.
fibn(K, I+1, N1+N2) ← I<K, fib(K, I, N1), fib(K, I-1, N2).
```

A query

```
?fibn(10, J, N).
```

will be supported correctly in both LDL and Prolog (modulo some syntactic de-sugaring). However, LDL will do that efficiently, by a single fixpoint loop with no duplicate work, while Prolog will be hopelessly inefficient since it recomputes the same Fibonacci numbers over and over again. Rewriting fibn, to ensure efficient execution in Prolog, is not a trivial matter. An even more dramatic example of the advantages of LDL in handling recursive predicates is offered by the non-linear ancestor example,

anc(X,Z) ← anc(X, Y) , anc(Y,Z).
anc(X,Y) ← parent(X,Y).

which, in Prolog, will never terminate, irrespective of the set of bindings used, while it is always safe to execute in LDL. (Also remember that the linear version of the ancestor of Figure 1 will not terminate in Prolog if the relation parent contains cycles, while it is safe in LDL. Moreover, while our examples emphasize database retrievals the fixpoint based computation of recursion works equally well when function symbols are present, as shown by the revzap example in Section 4.) In summary, LDL offers several advantages over Prolog with respect to recursive predicates. On the other hand, techniques such as appending lists through difference lists are not available in LDL, since solutions are always returned as (ground) data.

The banning of cuts from LDL yields significant differences with respect to Prolog. In most cases of common usage, the cut is simply replaced by the if-then-else construct available in LDL, producing programs that are easy to understand and efficient to implement [Okee]. In certain situations, however, the cut is used in Prolog to eliminate a computation that is unecessary because logical considerations lead to a single solution. Take, for instance the situation where the Prolog programmer is dealing with an existential goal, such as in the rule

q(X) :- r(Y),! , p(X).

Since the value of Y is unimportant, provided that it exists, the cut eliminates unecessary search for alternatives. As a second example, take the situation where the programmer is aware of constraints which will expedite the search. For instance to find the manager of an employee, the Prolog programmer can write

mg(Name, DpMgName) :-
    emp(Name, Dept), dept(Dept, DpMgName), ! .

Here the programmer is taking advantage of the logical constraints that an employee can only work in one department, and each department only has one manager, thus creating the following functional dependency: Name → DpMgName.

To deal with these situations, LDL has introduced a *choice predicate* [KrNa], which from a semantic viewpoint can be viewed as selecting an arbitrary element from a set of values in the minimal model; from an implementation viewpoint only one such value needs to be computed. Thus, for the existential situation, the LDL programmer will write

q(X) ← p(X), r(Y), choice((), Y).

The meaning of "choice((), Y)" is that, out of all possible Y-values in the

program's minimal model, any will do. Also observe that, because of the declarative semantics of choice [KRNa], the order of goals is unimportant*. The rule for finding the manager of an employee will be expressed by the LDL programmer as follows:

```
mg(Name, DpMgName) ←
        emp(Name, Dept), dept(Dept, DpMgName),
        choice((Name), (DpMgName)).
```

The syntax

```
choice((X), (Y))
```

denotes that the functional dependency X → Y can be assumed to hold. LDL will use this information to avoid computing several Y values for a given X value. Thus the answer returned from the rule above will only list one manager for each employee name. The benefits gained by the fact that choice is a declarative construct are illustrated by the following queries:

```
?mg(X, Y).
?mg(X, joe_boss).
```

While the LDL program remains valid and amenable to efficient implementation via re-compilation, the old Prolog program is no longer good for these new queries and the programs will have to be rewritten.

The next program contrasts the use of the if-then-else predicate with that of the choice construct, and illustrates a typical computation of set aggregates in LDL. The following procedure finds the sum X of all elements of a non-empty set S:

```
sum(S, X) ← if S={Y} then X=Y else
        partition(S, S1, S2), choice((S), (S1, S2)),
        sum(S1, X1), sum(S2,X2), X= X1 + X2.
```

The sum of the elements of a set S can be computed by partitioning it into two non-empty subsets and then adding the sum of the two. Because of the associativity and commutativity of sets, there is no point in considering all the possible dichotomies of S, since any pair will do —thus the use of the choice predicate.

The differences between programming in Prolog and LDL are not limited to those discussed here pertaining to recursive predicates and choice predicates. For instance, the powerful update and set constructs of LDL result in rather terse and expressive programs using these constructs [NaTs]. On the other hand, there are no metalevel facilities in LDL, although this issue is currently being pursued [KrNa].

---

* In this particular case, there is no need for the user to add a choice goal, since the LDL compiler is smart enough to add it automatically to the rule [RaBK].

Acknowledgments

LDL is the result of the shared efforts by many dedicated colleagues whom I am bound to by gratitude and friendship. In particular, I would like to recognize the contribution and dedication of the following persons: François Bancilhon, Catriel Beeri, Danette Chimenti, Ruben Gamboa, Charles Kellog, Paris Kanellakis, Ravi Krishnamurthy, Tony O'Hare, Kayliang Ong, Arshad Matin, Raghu Ramakrishnan, Shamim Naqvi, Domenico Saccà, Oded Shmueli, Leona Slepetis, Peter Song, Millie Villareal, Shalom Tsur, Carolyn West. I am also grateful to Manuel Hermenegildo and Roger Nasr for many stimulating technical discussions and comments on this paper.

References

[AhUl] Aho A. V. and J. Ullman, " Universality of Data Retrieval Languages," *Proc. POPL Conference,* San Antonio Tx, 1979.

[AiNa] Ait-Kaci H. and R. Nasr, "Le Fun: Logic, Equations, and Functions" *Procs. of IEEE Symposium on Logic Programming,* pp.17-23, 1987.

[ApBW] Apt, K., H. Blair, A. Walker, "Towards a Theory of Declarative Knowledge," *in Foundations of Deductive Databases and Logic Programming,* (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[BaBu] Bancilhon, F. and P. Buneman (eds.), "Workshop on Database Programming Languages," Roscoff, Finistere, France, Sept. 87.

[Ban] Bancilhon, F., "Naive Evaluation of Recursively defined Relations", *On Knowledge Base Management Systems,* (M. Brodie and J. Mylopoulos, eds.), Springer-Verlag, 1985.

[BaR] Balbin, I., K. Ramamohanarao, "A Differential Approach to Query Optimization in Recursive Deductive Databases", *Journal of Logic Programming,* Vol. 4, No. 2, pp. 259-262, Sept 1987.

[BaRa] Bancilhon, F., and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. ACM SIGMOD Int. Conference on Management of Data,* Washington, D.C., May 1986.

[BeRa] Beeri, C. and R. Ramakrishnan, "On the Power of Magic," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1987.

[Bet1] Beeri, et al., "Sets and Negation in a Logic Data Language (LDL1)", *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 269-283, 1987.

[Bet2] Beeri, et al., "Bound on the Propagation of Selection in Logic Programs", *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1987.

[BKBR] Beeri, C., P. Kanellakis, F. Bancilhon, R. Ramakrishnan, "Bound on the Propagation of Selection into Logic Programs", *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1987.

[BMSU] Bancilhon, F., D. Maier, Y. Sagiv, J. Ullman, "Magic sets and other strange ways to implement logic programs", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1986.

[Boc] Bocca, J., "On the Evaluation Strategy of Educe," *Proc. 1986 ACM-SIGMOD Conference on Management of Data,* pp. 368-378, 1986.

[CeGW] Ceri, S., G. Gottlob and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," *Expert Database Systems,* L. Kerschberg (ed.), Benjamin/Cummings, 1987.

[Ceta] Chimenti D. et al., "An Overview of the LDL System," *Database Engineering Bulletin,* Vol. 10, No. 4, pp. 52-62, 1987.

[Col] Colmerauer, "Equations and Inequations in Finite and Infinite Trees," *Proc. Int. Conf. on Fifth Generation Computer Systems,* pp. 85-99, ICOT, Tokyo, Japan, 1984.

[CoMa] Copeland, G. and Maier D., "Making SMALLTALK a Database System," *Proc. ACM SIGMOD Int. Conf. on Management of Data,* pp. 316-325, 1985.

[DaKV] Danforth, S.,S. Khoshafian and P. Valduriez, "FAD- A Database Programming Language. Rev 2", Submitted for publication.

[deSi] deMandreville C. and E. Simon, "Modelling Queries and Updates in Deductive Databases" *Proc. 1988 VLDB Conference,* Los Angeles, California, August 1988.

[GaDe] Gardarin, G. and C. deMandreville, "Evaluation of Database Recursive Logic Programs as Recursive Function Series," *Proc. ACM SIGMOD Int. Conference on Management of Data,* Washington, D.C., May 1986.

[GMN] Gallaire, H.,J. Minker and J.M. Nicolas,"Logic and Databases: a Deductive Approach," *Computer Surveys,* Vol. 16, No. 2, 1984.

[Har] Harel, D., "First-Order Dynamic Logic," *Lecture Notes in Computer Science,* (G. Goos and J. Hartmanis, eds.), Springer Verlag, 1979.

[HeNa] Henschen, L.J., Naqvi, S. A., "On compiling queries in recursive first-order databases", *JACM 31,* 1, 1984, pp. 47-85.

[ImNa] Imielinski, T. and S. Naqvi, "Explicit Control of Logic Programs Through Rule Algebra," *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 103-116, 1988.

[IoWo] Ioannidis, Y. E. and E. Wong, "Query Optimization by Simulated *Proc. ACM SIGMOD Int. Conf. on Management of Data,* 1987.

[JaCV] Jarke, M., J. Clifford and Y. Vassiliou, "An Optimizing Prolog Front End to a Relational Query System," *Proc. 1984 ACM-SIGMOD Conference on Management of Data,* pp. 296-306, 1986.

[KeOT] Kellog, C., A. O'Hare and L. Travis, "Optimizing the Rule Data Interface in a KMS," *Proc. 12th VLDB Conference,* Tokyo, Japan, 1986.

[KiLo] Kifer, M. and Lozinskii, E.L., "Filtering Data Flow in Deductive Databases," *ICDT'86,* Rome, Sept. 8-10, 1986.

[KoPa] Kolaitis G. P. and C.H. Papadimitriou, "Why Not Negation by Fixpoint?", *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 31-239, 1988.

[KuYo] Kunifji S., H. Yokota, "Prolog and Relational Databases for 5th Generation Computer Systems," in *Advances in Logic and Databases, Vol. 2* (Gallaire, Minker and Nicolas eds.), Plenum, New York, 1984.

[KrBZ] Krishnamurthy, R., H. Boral and C. Zaniolo, "Optimization of Non-Recursive Queries," *Proc. 12th VLDB,* Kyoto, Japan, 1986.

[KrN1] Krishnamurthy and S. Naqvi, "Non-Deterministic Choice in Datalog," *Proc. 3rd Int. Conf. on Data and Knowledge Bases,* June 27-30, Jerusalem, Israel.

[KrN2] Krishnamurthy and S. Naqvi, "Towards a Real Horn Clause Language," *Proc. 1988 VLDB Conference,* Los Angeles, California, August 1988.

[KrRS] Krishnamurthy, R. R. Ramakrishnan and O. Shmueli, "A Framework for Testing Safety and Effective Computability," *Proc. ACM SIGMOD Int. Conf. on Management of Data,* pp. 154-163, 1988.

[KrZa] Krishnamurthy, R. and C. Zaniolo, "Optimization in a Logic Based language for Knowledge and Data Intensive Applications," in *Advances in Database Technology, EDBT'88,* (Schmidt, Ceri and Misssikoff, Eds), pp. 16-33, Springer-Verlag 1988.

[Li] Li, D. "*A Prolog Database System,*" Research Institute Press, Letchworth, Hertfordshire, U.K., 1984

[Llo] Lloyd, J. W., *Foundations of Logic Programming,* Springer Verlag, (2nd Edition), 1987.

[Meta] Morris, K. et al. "YAWN! (Yet Another Window on Nail!), *Data Engineering,* Vol.10, No. 4, pp. 28-44, Dec. 1987.

[Nai] Naish, L., "Negation and Control in Prolog", *Lecture Notes in Computer Science 238,* Springer Verlag 1986.

[NaKr] Naqvi, S. and R. Krishnamurthy, "Semantics of Updates in logic Programming", *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 251-261, 1988.

[Naq] Naqvi, S. "A Logic for Negation in Database Systems," *in Foundations of Deductive Databases and Logic Programming,* (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[NaTR] Naish, J. A., A. Thom and K. Ramamohanarao, "Concurrent Database Updates in Prolog," *Proc. Fourth Int. Conference on Logic Programming,* Melbourne, Australia, 1987. pp. 178-195, 1987.

[NaTs] Naqvi, S. and S. Tsur, "A Logic Language for Data and Knowledge Bases," MCC Technical Report, 1988.

[Okee] O'keefe, R.A., "On the Treatment of Cuts in Prolog Source Level Tools," *Proc. Symposium on Logic Programming,* pp. 68-73, 1985.

[Prz] Przymusinski, T., "On the Semantics of Stratified Deductive Databases and Logic Programs", *in Foundations of Deductive Databases and Logic Programming,* (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[RaBK] Ramakrishnan, R., C. Beeri and Krishnamurthy, "Optimizing Existential Datalog Queries," *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 89-102, 1988.

[Rei] Reiter, R., "On closed world databases", in *Logic and Databases* (Gallaire, H., Minker, J., eds), Plenum, New York, 1978, pp. 55-76.

[Reta] Ramamohanarao, K. et al. "The NU-Prolog Deductive Database System" *Database Engineering Bulletin,* Vol. 10, No. 4, pp. 10-19, 1987.

[RLK] Rohmer, J., R. Lescouer and J.M. Kerisit,"The Alexander Method — A technique for the Processing of Recursive Axioms in Deductive Databases" *New Generation Computing,* Vol. 4, No. 3, pp. 273-287, 1986.

[SaZ1] Saccà, D., Zaniolo, C., "On the implementation of a simple class of logic queries for databases", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1986.

[SaZ2] Saccà, D., Zaniolo, C., "Implementation of Recursive Queries for a Data Language based on Pure Horn Logic," *Proc. Fourth Int. Conference on Logic Programming,* Melbourne, Australia, 1987.

[SaZ3] Saccà, D., Zaniolo, C., "The Generalized Counting Method for Recursive Logic Queries," *JTC, to appear,* (also *Proc. ICDT '86).*

[SaZ4] Saccà, D., Zaniolo, C., "Magic Counting Methods," *Proc. ACM SIGMOD Int. Conf. on Management of Data,* 1987.

[SaZ5] Saccà, D., Zaniolo, C., "Differential Fixpoint Methods and Stratification of Logic Programs," *Proc. 3rd Int. Conf. on Data and Knowledge Bases,* June 27-30, Jerusalem, Israel.

[Seta] Selinger, P.G. et al. "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Int. Conf. on Management of Data,* 1979.

[ShNa] Shmueli, O. and S. Naqvi, "Set Grouping and Layering in Horn Clause Programs," *Proc. of 4th Int. Conf. on Logic Programming,* pp. 152-177, 1987.

[ShTZ] Shmueli, O., S. Tsur and C. Zaniolo, "Rewriting of Rules Containing Set Terms in a Logic Data Language (LDL)," *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 15-28, 1988.

[SmGe] Smith, D.E. and M.R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence,* 26, pp. 171-185, 1985.

[TsZa] Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," *Proc. of 12th VLDB,* Tokyo, Japan, 1986.

[Ull] Ullman, J.D., *Database and Knowledge-Based Systems,* Computer Science Press, Rockville, Md., 1988.

[vEKo] van Emden, M.H., Kowalski, R., "The semantics of Predicate Logic as a Programming Language", *JACM 23,* 4, 1976, pp. 733-742.

[Vie] Vieille, L. "Recursive Axioms in Deductive Databases: the Query-Subquery Approach," *Proc. First Int. Conference on Expert Database Systems,* Charleston, S.C., 1986.

[Vil] Villareal, E., "Evaluation of an O(N**2) Method for Query Optimization," MS Thesis, Department of Computer Science, University of Texas at Austin, 1987.

[War] Warren, D.H.D., "An Abstract Prolog Instruction Set," Tech. Note 309, AI Center, Computer Science and Technology Div., SRI, 1983.

[Zan1] Zaniolo, C. "Prolog: a database query language for all seasons," in *Expert Database Systems, Proc. of the First Int. Workshop* L. Kerschberg (ed.), Benjamin/Cummings, 1986.

[Zan2] Zaniolo, C. "The Representation and Deductive Retrieval of Complex Objects," *Proc. 11-th VLDB,* pp. 459-469, 1985.

[Zan3] Zaniolo, C. "Safety and Compilation of Non-Recursive Horn Clauses," *Proc. First Int. Conference on Expert Database Systems,* L. Kerschberg (ed.), Benjamin/Cummings, 1986.