Efficient Processing of Declarative Rule-Based Languages for Databases

Carlo Zaniolo
MCC
Austin, Texas, USA

Abstract

In recent years, Deductive Databases have progressed from a subject of theoretical interest to an emerging technology area of significant commercial potential. The two main catalysts for progress have been a demand for advanced database applications and a rapid maturation of enabling technology. Thus, the area of Deductive Databases has now progressed beyond its initial Prolog-oriented beginnings and produced logic-based languages, architectures and systems that support a declarative expression on knowledge through rules and their efficient processing on large databases. In this paper, we review the key concepts behind deductive databases, including language constructs, semantics issues, implementation techniques, architectures and prototypes. Then, we discuss key application areas driving the development of this technology, and current research directions on systems and theory.

1 Introduction

Deductive Databases support *queries, reasoning*, and *application development* on databases through a *declarative rule-based* languages. The practical motivations for this novel technology is the emergence of a new wave of database applications that are not supported well current technology. Examples include:

1. *Computer aided design and manufacturing systems.*
2. *Scientific database applications:* Examples include studies of chemical structures, genomic data, and analysis of satellite data.
3. *Knowledge Mining and Data Dredging:* This require support for browsing and complex ad-hoc queries on large databases [Ts2]. For example, researchers need to search through medical histories to validate hypotheses about possible causes of diseases, and an airlines want to maximize yields resulting from schedules and fare structure.

In addition to the traditional requirements of databases (such as integrity, sharing and recovery), these new applications demand complex structures, recursively defined objects, high-level languages and rules. For example, a VLSI CAD system typically allows the definitions of "cells," which are designs having other cells as subparts. Operations on such a design often begin by expanding out the design, say, to create a checkplot. The expansion must be carried on to arbitrary depth, so a recursive logic program is appropriately used to define the operation of cell expansion.

These new applications cannot be supported effectively by commercial database systems, including relational systems exemplified by SQL. Furthermore, even current applications suffer from the limited power of current DBMS, where languages such as SQL can only access and modify data in limited ways; thus database applications are now written in a conventional language with intermixed query language calls. But since the nonprocedural, set-oriented computational model of SQL is so different from that of procedural languages, and because of incompatible data types, an "impedance mismatch" occurs that hinders application development and causes expensive run-time conversions. This problem is particularly acute in applications such as the Bill Of Materials dealing with arbitrary structures of unlimited depth. Thus, it has become generally accepted that for applications at the frontier we need a single, computationally complete language that answers the needs previously discussed and serves both as a *query language* and as a *general-purpose host language*.

Object-oriented systems, where the database is closely integrated with languages such as Smalltalk or C++, address some of the previous requirements, and support useful concepts, such as object-identity and a rich type structure with inheritance of properties from types to their subtypes. However, object-oriented systems lose an important advantage that relational languages have: relational languages are declarative and logic-based. Declarative languages provide the ability to express what one wants, and leave for invention by the system substantial portions of the algorithm required to meet the request. This ability is essential for ease of use, data independence and code reusability.

Thus, deductive databases take the declarative approach in addressing those requirements: they provide a *declarative, logic-based* language for expressing *queries, reasoning*, and complex *applications* on databases.

Interest in the area began in the early seventies with the establishment of the theoretical foundations for the field [GMN], and experimental activity limited to few groundbreaking experiments [Kell]. A new generation of powerful rule-based languages for expert systems applications commanded great attention in the 80's. Among these, Prolog is of particular interest, because it is based on extensions of Horn-clause logic; Horn clauses are a close relative of relational calculus, which provides the semantic underpinning for relational query languages such as SQL. This similarity has led to considerable work at building a deductive database system, either, by extending Prolog with database capabilities or by coupling it with relational DBMSs [CGT]. While these experiments have been successful in producing powerful systems, they have also revealed several

problems that stand in the way of complete integration. Some of these problems follow from the general difficulty of marrying DBMSs with programming languages, others are specific to Prolog.

Some of Prolog's limitations, such as the lack of schemas and of secondary-storage based persistence, can be corrected through suitable extensions; others are so deeply engrained in Prolog's semantics and enabling technology that they are very difficult to overcome. For instance, the cornerstone of Prolog is SLD-resolution according to a left-to-right, depth-first execution order. This powerful mechanism provides an efficient implementation for Horn-clause logic and an operational semantics to the many non-logic based constructs—such as updates, cuts and meta-level primitives—that were added to the language for expressive power. But the dependence of Prolog, and its enabling technology, on SLD-resolution present serious drawbacks from a database viewpoint:

- Prolog's rigid execution model corresponds to a navigational query execution strategy; thus, it compromises data independence and query optimization that build upon the non-navigational nature of relational query languages.
- This rigid semantics is incompatible with several relational database concepts—in particular with the notions of database updates and transactions. For instance, in the style of many AI systems, Prolog's update constructs (i.e., assert and retract) are powerful but unruly, inasmuch as it can modify both the data and the program. Furthermore, none of the nine different semantics for updates in Prolog counted so far [Moss], are compatible with that of the relational data model. Indeed, the snapshot-based semantics of relational databases is incompatible with Prolog's execution model, which is instead oriented toward pipelined execution [KNZ]. Supporting the notion of transactions, which is totally alien to Prolog, compounds these problems.
- The efficiency of Prolog's execution model is predicated upon the use of main memory. Indeed, all current Prolog implementations [WAM] rely on pointers, stacks and full unification algorithms, which are not well-suited to a secondary store-based implementation.

Thus, several research projects were undertaken aiming to achieve a complete and harmonious integration of logic and databases; for the reasons just mentioned, these projects have rejected Prolog's SLD-based semantics and implementation technology, but retained Horn clauses with their rule-oriented syntax. This line of research has produced new languages and systems that combine the database functionality and non-procedurality of relational systems, with Prolog's reasoning and symbolic manipulation capability. A new implementation technology was developed for these languages (using extensions of relational DBMSs technology) to ensure their efficient support on, both, main memory and secondary store. Among the several prototypes proposed [Meta,KiMS] we will base our discussion on the $\mathcal{LDL}$ system [Ceta], due to the level of maturity that it has reached and the author's familiarity with the system. Fully integrated deductive database systems have the following distinguished traits:

- Support for all database essentials. There is a clear notion of a (time-varying) database separated from the (time-invariant) rule-based program. The database is described by a schema with unique key constraints declarations, and explicit indexing information. Schemas are thus the vehicle for integrity constraint declaration, for accessing both internal relations and external relations from SQL databases. The notions of recovery and database transactions are thus deeply engrained in languages such as $\mathcal{LDL}$ [KNZ].

- A semantics that is database-oriented, declarative, and rigorous, as illustrated by the following points:

  1. *Database Orientation:* For instance, a snapshot-based semantics is used for updates in $\mathcal{LDL}$, combined with full support for the concept of database transactions. Other concepts that are directly derived and extended from relational databases include all-answer solutions; duplicate control; sets, nested relations; and the ability to enforce key constraints and functional dependencies in derived relations (via the choice construct).
  2. *Declarative Language:* As discussed in the next section, these systems come closer to implementing the full declarative semantics of Horn clauses, by supporting both forward chaining and backward chaining execution strategies, under automatic system control [UlZa]. Thus, several applications, e.g., those involving non-linear rules or cyclic graphs, are much simpler to write in $\mathcal{LDL}$ than in Prolog [UlZa]. The notion of a query optimizer is also part of these systems, for compatibility with relational systems, better data independence, and enhanced program reusability. Finally, the declarative semantics is extended beyond the Horn Clauses to include stratified negation, grouping and non-deterministic pruning (thus eliminating Prolog's cut) [NaTs].
  3. *Rigorous Semantics:* $\mathcal{LDL}$'s well-documented semantics [NaTs], is the result of a systematic effort to ferret out any ambiguity from both the declarative and the imperative aspects of the language. For instance, in dealing with logical constructs, such as negation and set-grouping, non-stratified programs are disallowed due to the lack a model-theoretic semantics for some of these programs. For imperative constructs, several restrictions are enforced upon programs with updates, such as disallowing updates in disjunctive goals and prohibiting unfailing goals after updates. The objective of these restrictions is to simplify and structure these programs along with their compilation. As a result, $\mathcal{LDL}$ programs with updates must be structured in a precise way—a discipline that requires some learning, but also enhances the value of resulting code as a vehicle for rigorous and complete specifications.

- An implementation technology that is database-oriented, and, in fact, represents an extension to the compiler/optimizer technology of relational systems. Thus SLD-resolution and unification are respectively replaced with fixpoint computation and matching, which because of their simpler nature can be supported well in secondary as well as in primary storage [Ullm,Ceta]. Furthermore, declarative

set-oriented semantics, makes it implementable using an assortment of alternative execution models and strategies— including translation to relational algebra— thus expanding on the opportunities for query optimization of relational systems.

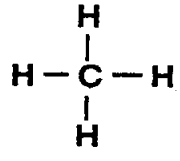
We will next review the most salient features of these systems in terms of languages, applications and architectures.

2 Languages

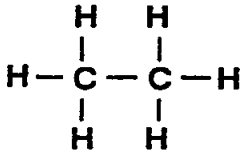
Logic-based languages for databases include three kinds of constructs:

1. Horn-clause based constructs
2. Non-monotonic logic-based constructs (such as negation, sets and choice operators)
3. imperative constructs (such as updates and I/O)

A language such as $\mathcal{LDL}$ shares with Prolog Horn-clause based constructs above, but not the remaining two. There are significant differences even with respect to Horn Clauses, as illustrated by the fact that in deductive databases programs are less dependent on a particular execution model, such as forward-chaining or backward-chaining. A Prolog programmer can only write rules that work with backward chaining; an OPS5 programmer can only write rules that work in a forward chaining mode. By contrast, systems such as $\mathcal{LDL}$ [NaTs] and NAIL! [Meta], select the proper inference mode automatically, enabling the user to focus on the logical correctness of the rules rather than on the underlying execution strategy. This point is better illustrated by an example. A methane molecule consists of a carbon atom linked with four hydrogen atoms. An ethane molecule can be constructed by replacing any H of a methane with a carbon with three Hs. Therefore, the respective structures of methane and ethane molecules are as follows:



methane



ethane

More complex alkanes can then be obtained inductively, in the same way: i.e., by replacing an H of a simpler alkane by a carbon with three Hs.

We can now define alkanes using Horn clauses. A methane molecule will be represented by a complex term, `carb(h, h, h)`, and an ethane molecule by `carb(h, h,`

carb(h, h, h))(thus we implicitly assume the presence of an additional h, the root of our tree). In general, alkane molecules can be inductively defined as follows:

```
all_mol(h, 0, Max).
all_mol(carb(M1, M2, M3), N, Max) ←
                    all_mol(M1, N1, Max),
                    all_mol(M2, N2, Max),
                    all_mol(M3, N3, Max),
                    N= N1+N2+N3+1, N <= Max.
```

In addition to defining alkanes of increasing complexity, these non-linear recursive rules count the carbons in the molecules, ensuring finiteness in their size and number by checking that the tally of carbons never exceeds Max.

This alkane definition can be used in different ways. For example, to generate all molecules with no more than four carbons, one can write:

```
? all_mol(Mol,Cs, 4).
```

To generate all molecules with exactly four carbons one will write:

```
? all_mol(Mol, 4, 4).
```

Furthermore, if the relation alk(Name, Str) associates the names of alkanes with their structure, then the following rule will compute the number of carbons for an alkane given its name (assume that 10000 is a large enough number for all molecules to have a lower carbon complexity).

```
find(Name, Cs) ← alk(Name, Str), all_mol(Str, Cs , 10000).
```

The first two examples can be supported only through a forward chaining computation, which, in turn, translates naturally into the least-fixpoint computation that defines the model-theoretic based semantics of recursive Horn clause programs [NaTs]. The least fixpoint computation amounts to an iterative procedure, where partial results are added to a relation until a steady state is reached.

Therefore, deductive databases support well the first two examples via forward chaining, while Prolog and other backward chaining systems would flounder. In the last example, however, the first argument, Str, of all_mol is bound to the values generated by the predicate alk. Thus a computation such as Prolog's backward chaining, which recursively propagates these bindings, is significantly more efficient than forward chaining. Now, deductive databases solve this problem equally well, by using techniques such as the *Magic Set Method*, or the *Counting Method* that simulate backward chaining through a pair of coupled fixpoint computations [Ullm].

Since fixpoint computations check newly generated values against the set of previous values, cycles are handled automatically. This is a most useful feature since cyclic graphs are often stored in the database; furthermore, derived relations can also be cyclic. In our alkane example, for instance, there are many equivalent representations for the same alkane. To generate them, equivalence-preserving operations, such as rotation and permutation on the molecules, are used—but repeated applications of these operations bring back the initial structure. In deductive databases, there is no need to carry around a bag containing all previous solutions, since cycles are detected and handled efficiently by the system.

3 Non-Monotonic Constructs

The declarative semantics and programming paradigm of deductive databases extend beyond Horn clause programming, to include non-monotonic logic-based constructs, such as negation, sets and choice operators [NaTs]. In fact, this line of research has much contributed to the advancement of semantics and implementation techniques for non-monotonic logic constructs. For instance, deductive databases support efficiently stratified negation [NaTs], and has a rigorous semantics based on the concept of perfect models [Prz]. While stratified negation is more powerful than negation-by-failure provided by Prolog, many applications require unstratified negation (and set aggregation). Research in this area has produced elegant concepts, such as well-founded models [VGRS] and stable models, which provide very general declarative semantics for logic programs with negation [GeLi]. But efficient implementations for such semantics remains, in general, an open problem.

Significant progress was accomplished on non-deterministic pruning constructs, for which declarative logic-based semantics and efficient implementation techniques were developed [SaZa,GPSZ]. A construct called `choice` was introduced [KrNa], where a semantics based on functional dependency constraints was proposed. This semantics was then revised and extended using the notion of stable models [SaZa].

To illustrate these concepts, consider for instance the following example where *takes* denotes students taking courses.

a_st(St,Crs) ← *takes(St,Crs), choice((Crs),(St)).*
takes(andy,engl).
takes(ann, math).
takes(mark,engl).
takes(mark,math).

The choice goal in the first rule specifies that the *a_st* predicate symbol must associate exactly one student to each course. Thus the functional dependency $Crs \rightarrow St$ holds in the model defining the meaning of this program. A program with equivalent meaning can be defined using negation as follows:

$a\_st(St,Crs) \leftarrow takes(St,Crs),\ chosen(Crs,St).$
$chosen(Crs,St) \leftarrow takes(St,Crs), \neg\ diffChoice(Crs,St).$
$diffChoice(Crs,St) \leftarrow chosen(Crs,\ \overline{St}\ ), St \neq \overline{St}.$
$takes(andy,engl).$
$takes(ann,\ math).$
$takes(mark,engl).$
$takes(mark,math).$

This program with negation has stable model semantics, where the non-determinism is captured by the presence of alternative stable models. The stable-model semantics for the particular case of `choice` programs is also amenable to efficient implementation [GPSZ].

4 Architectures

The key implementation problems for Deductive Databases pertain to finding efficient executions for the given set of rules and query. For this purpose, Deductive Database systems perform a global analysis of rules—in contrast to Prolog compilers, which are normally based on local rule analysis. A global analysis is performed at compile time, using suitable representations such as the Rule/Goal graph [Ullm] or the predicate connection graph [Ceta]. Its cornerstone is the notion of bound arguments and free arguments of predicates. For a general idea of this global analysis is performed, consider the following example:

```
usanc(X, Y) <- anc(X,Y), born(Y, usa).

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

Thus, the last two rules supply the recursive definition of ancestors (parents of an ancestors are themselves ancestors) and the the first rule choses the ancestors of a given `X` that were born in the USA (lower case is used for constants, and upper case for variables). Then a query such as

```
? usanc(mark, Y).
```

defines the following pattern:

$$\texttt{usanc}^{\texttt{bf}}$$

The superscript bf is an *adornment* denoting the fact that the first argument is bound and the second is not.

The global analysis is next applied to determine how the adornments of the query goal can be propagated down to the rest of the rule set. By unifying the query goal with the head of the usanc rule, we obtain the adorned rule:

$$\mathtt{usanc}^{\mathtt{bf}} < -\mathtt{anc}^{\mathtt{bf}}, \mathtt{born}^{\mathtt{bb}}.$$

This adornment assumes that the first argument of born is bound by the second argument of anc according to a sideway information passing principle (SIP) [Ullm]. The next question to arise is whether the recursive goal $\mathtt{anc}^{\mathtt{bf}}$ is supportable. The analysis of the anc rules yields the following adorned rules (assuming a left-to-right SIP):

$$\mathtt{anc}^{\mathtt{bf}} < -\mathtt{parent}^{\mathtt{bf}}.$$
$$\mathtt{anc}^{\mathtt{bf}} < -\mathtt{parent}^{\mathtt{bf}}, \mathtt{anc}^{\mathtt{bf}}.$$

The analysis is now complete, since the adornment of the anc goal in the tail is the same as that in the head. Assuming that born and parent are database predicates, the given adornments can easily be implemented through a search taking advantage of the bound first argument in parent and both bound arguments in born. The recursive predicate anc can also be solved efficiently: in fact, a further analysis indicates that the recursive rule is left-linear [Ullm] and that the given adornment can, after some rewriting of the rules, be supported by a single-fixpoint computation [Ullm]. When the recursive predicate cannot be supported through a single fixpoint, other methods are used, including the counting method, and the very general magic set method [Ullm].

Figure 1 describes the architecture of the $\mathcal{LDL}$ system. The first operation to be performed once a query form is given (a query form is a query template with an indication of bound/free arguments) is to propagate constants into recursive rules and to extract the subset of rules relevant to this particular query. By examining alternative goal orderings, execution modes, and methods for supporting recursion the optimizer finds a safe strategy, which minimizes a cost estimate. For rules where all goals refer to database relations, the optimizer behaves like a relational system. The Enhancer's task, is to apply the proper recursive method by rewriting the original rules. A rule rewriting approach is also used to support the idempotence and commutativity properties of set terms. Since recursion is implemented by fixpoint iterations, and only matching is needed at execution time, the abstract target machine and code can be greatly simplified, with respect to that of Prolog [WAM]; thus, it can also be based on simple extensions to relational algebra.

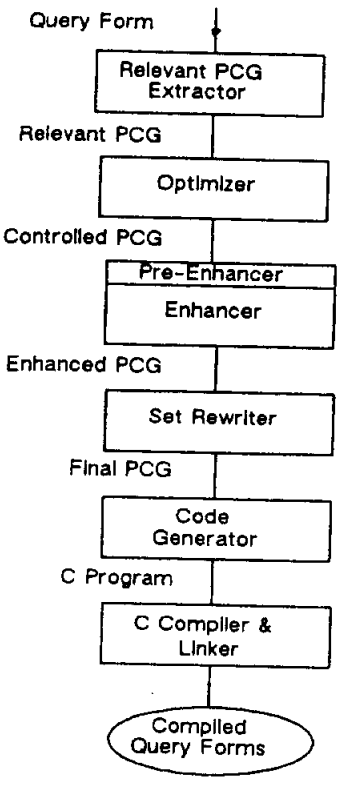


Figure 1. Architecture of the $\mathcal{CDL}$ System

For instance, the first (limited) $\mathcal{CDL}$ prototype generated code for an intermediate relational-algebra language for a parallel database machine. The current prototype is based on single-tuple get-next interface designed for both main-memory and secondary store. The single-tuple interface supplies various opportunities for intelligent backtracking and existential variables optimization, exploited by the compiler to obtain good performance from the object code [Meta]. The intermediate object code is actually C, to support portability and a open architecture.

Other experimental systems differ in several ways from the architecture of Figure 1. For instance, NAIL! uses a relational algebra-based intermediate code, and employs *capture rules*, rather than cost-prediction based optimization, to drive the selection of a proper execution strategy [Ceta].

5 Applications

The unique advantages offered by Deductive Databases in several applications areas are well-documented and demonstrated by various pilot applications. These areas range from traditional ones, such as computer-aided manufacturing applications, which presently suffer because of the inability of SQL to support recursive queries and rules, to

new scientific applications, such as those connected with the burgeoning areas of molecular biology [Ts1]. Because of space limitations, we will discuss only data dredging and enterprise modeling.

Data Dredging: This term denotes an emerging computational paradigm which supports "knowledge extraction" from, and the "discovery process" on the ever-growing repository of stored data [Ts2]. This usage of databases—in the past primarily associated with the intelligence community— is now becoming pervasive in medicine and science. Data Dredging is also an increasingly common practice of such such business applications as selective marketing and yield-management [Hopp].

The source of the data is typically a large volume of low-level records, collected from measurements and monitoring of empirical processes, intelligence operations and businesses. The problem is how to use this data to verify certain conjectures and to help refine or formulate hypotheses. Typically, the level of abstraction at hypotheses are formulated is much higher than that at which the data was collected. Thus, an iterative approach is needed, as follows:

1. Formulate hypothesis or concept.
2. Translate the concept into an executable definition (e.g., a rule-set and query).
3. Execute the query against the given data and observe the results.
4. If the results fully confirm the hypothesis, then exit; otherwise, modify the initial hypothesis and repeat these steps.

Obviously, the decision to exit the process is subjective and upon the analyst or researcher who is carrying out the study. At this stage he or she may have decided either that the concept is now adequately finalized and substantiated, or that the data does not support the initial conjecture and should be abandoned or tried out with different data. While in principle, this procedure could be carried out using any programming language, the key to the experiment's practicality and timeliness hinges upon the ability to complete it within limited time and effort. Thanks to their ability of quickly formulating very sophisticated queries and ruled-based decisions on large volumes of data, deductive databases are an ideal tool for data dredging. Our experience in developing such applications with $\mathcal{LDL}$ also suggests that its open architecture is important in this process, inasmuch as, for example, a number of low-level, computation intensive tasks (such as filtering and preprocessing) must be used in the high-level, rule-driven discovery process.

Enterprise Modeling: The ability to model the data and the procedures of a business enterprise is key to the successful development of information systems. Some of the advantages of a deductive database environment in this respect were outlined in

the introduction; these advantages were confirmed during the one-year field study described in [Aeta]. This study reports on the experience of using the $\mathcal{LDL}$ prototype in conjunction with a structured-design methodology called *POS* (Process, Object and State) [].

A key idea of the POS methodology is that of using the E/R framework for modeling both dynamic and static aspects of the enterprise. By using the notions of aggregation and abstraction within the E/R framework, to capture what has traditionally been thought of as derived data, the E/R model can specify most of the processing associated with a specific problem domain. This allows the capture of both data modeling and process modeling within one framework, thus eliminating the need for additional formalisms (such as workflow diagrams) in the final specifications. Furthermore, when a deductive computing environment is used, both the traditional generalization structures and the less-often-used aggregation structures can be directly encoded in a rule-based description, yielding executable specifications that are well-structured, easy to read and have a formal semantics.

This basic approach was tested in a case study, where a a simplified information system was represented for the automobile registration authority (i.e., Department of Motor Vehicles). This information system involves the modeling of a set of entities (such as manufactures, owners, garages, and motor vehicles of various types) and a set of events or transactions, (such as the registration of various entities and the purchase and destruction of a motor vehicle). Several constraints must be enforced, including uniqueness, existence and cardinality of entities, and restrictions of parties qualified by law to partake in different transaction types. Specific applications to be supported by such an information system include:

- Knowing who is, or was, the registered owner of a vehicle at any time from its construction to its destruction
- Monitoring compliance with certain laws, such as those pertaining to fuel consumption and transfer of ownership

In an informal study, also including a comparison with alternative prototyping frameworks, $\mathcal{LDL}$ proved very effective and desirable, in terms of naturalness of coding, terseness, and readability of the resulting programs. A larger study is now in progress to determine the scalability of these benefits to applications in the large, and to further evaluate the following points:

- Use of $\mathcal{LDL}$ to validate large specifications
- Feasibility of an order of magnitude code compression over 3rd generation languages
- Shift in efforts from coders to requirements specifiers
- An increased scope of data management and decreased scope of application development organizations

6 Future Directions

By enabling the development of several pilot applications, the first generation of deductive database systems has proved the viability this very new technology and demonstrated its practical potential. Yet, these experiences have also revealed the need for several improvements and extensions. For this reason, and to take advantage of more recent technical advances, work is now in progress toward the next generation of integrated systems. These are expected to advance the state of the art in three major ways:

- *Correcting the limitations of current systems.* For, instance, the effectiveness of the current $\mathcal{LDL}$ system is hampered by the absence of good debugging facilities. Furthermore, while compiled $\mathcal{LDL}$ programs execute fast (outperforming commercial Prolog systems on data-intensive applications [Ceta]), the compilation of these programs is too slow. The new prototype under implementation, (called the $\mathcal{LDL}$++ System) will solve these problems by compiling into intermediate abstract machine code.

- *Reinforcing the strengths of current systems.* Several uses were found for $\mathcal{LDL}$'s flexible interface to external databases—as well as for its open architecture which allows the incorporation of external routines and data. These facilities will be greatly expanded in the $\mathcal{LDL}$++ prototype. On the other hand, the NAIL!/GLUE project is now pursuing a better amalgam with the procedural world through a closely coupled procedural shell.

- *Incorporating recent advances in theory of logic-based languages.* For instance, our understanding of non-monotonic logic has progressed to the point where a limited use of constructs such as negation and choice can now be allowed in recursion—thus entailing the writing of simpler and more expressive programs. Recently, there has also been significant progress on modeling and supporting objects in logic-based systems. These advances will be included in the next generation of deductive database systems.

References

[Aeta] Ackley, D., et al. "System Analysis for Deductive Database Environments: an Enhanced role for Aggregate Entities," *Procs. 9th Int. Conference on Entity-Relationship Approach*, Lausanne, CH, Oct. 8-10, 1990.

[Ceta] Chimenti, D. et al., "The $\mathcal{LDL}$ System Prototype," *IEEE Journal on Data and Knowledge Engineering*, March 1990.

[Hopp] Hopper, D.E., "Rattling SABRE—New Ways to Compete on Information," *Harvard Business Review*, May-June 1990, pp. 118-125.

[CGT] Ceri, S., G. Gottlob and L. Tanca, "Logic Programming and Deductive Databases," Springer-Verlag, 1989.

[DM89] "The Rapid Prototyping Conundrum", DATAMATION, June 1989.

[Gane] Gane, C. "Rapid System Development," Prentice Hall, 1989.

[GeLi] Gelfond, M., and Lifschitz, V., "The stable model semantics for logic programming", Proc. 5th Int. Conf. and Symp. on Logic Programming, MIT Press, pp. 1070-1080, 1988.

[GPSZ] Giannotti, F., D. Pedreschi, Saccà, D., and Zaniolo, C., "Non-Determinism in Deductive Databases," MCC Technical Report, STP-LD-003-91.

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

[Kell] Kellogg, C., "A Practical Amalgam of Knowledge and Data Base Technology" Proc. of AAAI Conference, Pittsburg, Pa., 1982.

[KiMS] Kiernan, G., C. de Maindreville, and E. Simon "Making Deductive Database a Practical Technology: a step forward," Proc. 1990 ACM-SIGMOD Conference on Management of Data, pp. 237-246.

[KNZ] Krishnamurthy, S. Naqvi and Zaniolo, "Database Transactions in $\mathcal{LDL}$", *Proc. Logic Programming North American Conference*, pp. 795-830, MIT Press, 1989.

[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.

[KrNa] Krishnamurthy, R., and Naqvi, S.A., "Non Deterministic Choice in Datalog", *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, Morgan Kaufmann Pub., Los Altos, pp. 416-424, 1988.

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

[Moss] Moss, C., "Cut and Paste—defining the Impure Primitives of Prolog", *Proc. Third Int. Conference on Logic Programming*, London, July 1986, pp. 686-694.

[NaTs] S. A. Naqvi, S. Tsur *"A Logical Language for Data and Knowledge Bases"*, W. H. Freeman, 1989.

[Prz] Przymusinski, T.C., "On the Declarative and Procedural Semantics of Deductive Databases and Logic Programs", in *Foundations of Deductive Databases and Logic Programming*, (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987, pp. 193-216.

[SaZa] Saccà, D., and Zaniolo, C., "Stable models and non determinism in logic programs with negation", Proc. 9th, ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 205-218, 1990.

[Ts1] Tsur S., 'Applications of Deductive Database Systems," *Proc. IEEE COMCON Spring '90 Conf.*, San Francisco, Feb 26-March 2.

[Ts2] Tsur S., "Data Dredging," *Data Engineering*, Vol. 13, No. 4, IEEE Computer Society, Dec. 90.

[Ullm] Ullman, J.D., *"Database and Knowledge-Based Systems*, Vols. I and II, Computer Science Press, Rockville, Md., 1989.

[UlZa] Ullman, J. and C. Zaniolo, "Deductive Databases, Achievements and Future Directions," *SIGMOD Record*, pp. 77-83, Vol. 19, No. 4, ACM Press, Dec. 1990.

[VGRS] Van Gelder, A., Ross, K., Schlipf, J.S., "Unfounded Sets and Well-Founded Semantics for General Logic Programs", *ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, March 1988, pp. 221-230.

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

[Zani] Zaniolo, C. "Object Identity and Inheritance in Deductive Databases: an Evolutionary Approach," *Proc. 1st Int. Conf. on Deductive and O-O Databases*, Dec. 4-6, 1989, Kyoto, Japan.