Compiling Horn-Clause Rules in IBM's Business System 12 an Early Experiment in Declarativeness

Ghica van Emde Boas - Lubsen[1] and Peter van Emde Boas[2]

[1] IBM, the Netherlands, PO Box 24, 1420 AA Uithoorn; emdeboas@nl.ibm.com
[2] ILLC - WINS, University of Amsterdam,
Plantage Muidergracht 24, 1018 TV Amsterdam; peter@wins.uva.nl

Abstract. The tight connection which exists between the fragment of Prolog now known by the name Datalog [28] and the various calculi and algebras for Relational Database Systems was observed at several places in the late 70-ies and early 80-ies. The problem was to make this idea operational and to build a system which implemented it. Such systems today are known as *Deductive Databases*.
We describe the history of a hardly known project from the mid 80-ies where a prototype realizing this goal was produced[1]. We explain why the Relational Database system called *Business System 12*, developed by IBM in the Netherlands, and which became operational in 1983, turned out to provide the right functionality. We also indicate how this project influenced subsequent projects aimed at enhancing the degree of declarativeness in interfaces with database systems.

What we understand we can formalize; what is formalizable can be automated, and you will discover that someone has built it.

[1] This report is an extended version of our similarly named invited paper which will be presented at the 1998 Silver Jubilee SOFSEM meeting, held at Jasná, Slovakia, November 21-27 1998, and which will appear in the Springer LNCS series.

1 The Great Idea

Deductive Databases arose in the 1980-ies out of the confluence of three technologies which came to age by the end of the 1970-ies. First, by 1980 the *Relational Database* had become the preferred model of database technology. Secondly, due to the initial success of *Expert Systems*, Artificial Intelligence was returned to grace. Finally, with the proliferation of the language Prolog the concept of *Logic Programming* became a prominent activity in Artificial intelligence, if not in computer science in general.

People observed that, notwithstanding the substantial differences in conceptual frameworks, these three approaches to conceptual information processing were storing and combining information in similar ways. Thus the idea arose to substitute the computational strategies from one field for the slower methods used in another. For example one might use the database engine from a Relational Database system in order to compute the results which would be produced by backtracking in Prolog.

Such a cross fertilization of technologies held great promises for all fields involved. Databases were much more powerful when crunching raw data than interpretation based Prolog systems. Secondary benefits from Database technology, like security, sharing and persistence, would become available to AI almost free. On the other hand database technology would benefit from the *declarativeness* of Prolog, where one can introduce new concepts and use them in the same way as concepts stored in the system directly. It would also provide a monolingual alternative for the rather heterogeneous ad hoc formalisms used for providing structural information about data and constraints to the database. That at the *procedural level* the match would be less than perfect (e.g., the order in which answers would be produced would be different in the two worlds), was something one could live with.

In this section we give our perspective on the state of the art in Databases, Ai and Logc programming around 1982. With a familiar example we show how relations are expressed in the three formalisms. Finally we explain our *compilation based* approach to building a Deductive Database. In section 2 we describe our project from 1984, where a fragment of Prolog indeed was compiled onto an existing database system. How the functionality of our prototype can be extended is discussed in the third and final section.

The reader should note that a number of terms naming and identifying concepts relevant to our project (terms like *Deductive Databases* and *Datalog*) still had to be invented in 1984. Using these phrases in our paper therefore is in fact an anachronism.[2]

1.1 The Theater

On the left side of the scene a team of engineers is programming a huge mainframe machine; on the other side a group of logicians is writing strange looking formulas on a blackboard....

[2] Evidently the actual developments in the fields of Databases, Logic Programming and Artificial Intelligence go far beyond the small part which the authors observed and participated in. The observations and opinions expressed in this paper therefore don't pretend to present a complete view on history, but rather to reflect the insights and positions held by the authors during these developments.

Relational Databases The main cause for the success of the Relational Database model as proposed by Codd [5] is its clean and conceptual semantics over a well understood mathematical model: relations are subsets of Cartesian products of Domains. Relations can be used both for describing *entities (objects)* by listing their characteristic properties as a tuple of values, and conceptual *relationships* between such entities. The conceptual world from the system designer (as represented in an entity-relationship scheme [4]) can be mapped to a Relational Database scheme, consisting of domains, attributes, tables and various constraints. This mapping is well understood and forms a basic ingredient of elementary database courses.

Relations in mathematics are used to represent the meaning of *predicates* in *first order predicate logic* (standard Tarskian semantics), so it is natural to use first order logic as a language for talking about the contents of databases. By 1980 the appropriate fragment of first order predicate logic had been described. This fragment today is known in various syntactic dialects by the names *domain calculus* or *tuple calculus*. The core of the prominent language SQL is a dialect of tuple calculus.

These calculi can be used to give a *declarative* description of *derived relations* in terms of *primitive relations* stored in the database[3]. Derived relations are evaluated by subjecting the primitive relations to suitable *algebraic operations* which live in the mathematical structure known by the name *relational algebra*. Well known relational operations are, *select, project, union, product, (natural) join* and *difference*. However, a single description may correspond to a large variety of equivalent expressions in the relational algebra, all yielding the same result, but with possibly different processing costs. It is the task of a *query optimizer* to produce, for a given logical or algebraic expression, an equivalent one with optimal processing costs.

An important shortcoming of the Relational Database model is the lack of abstraction mechanisms at the level of domains: what for one observer is a primitive value may turn out to be a composed structured object for another. In the relational model attribute values are selected from domains consisting of atomic values only. The need for *composite structured values* has resulted in the *nested* relational model. Computer Industry however has moved in a different direction: in the world of *Object Orientation* composite structured objects are the norm, and by making them persistent the *Object Oriented Database* is obtained. However, by taking this road all the secondary functionality of databases had to be recreated, and along the way the nice features of the relational model, particularly those related to its clean mathematical semantics and *declarativeness*, where lost. *Object-Relational Databases* [27] represent an attempt to preserve the best of both worlds.

These semantic issues on how to represent information and how to express queries, should not make us forget the prime purpose of a database system: *providing a secure, stable and persistent environment where a community of users may concurrently access, query and update large volumes of shared data*. Database technology primarily is concerned with the technological problems of realizing these "secondary" features.

Artificial Intelligence and Logic Programming In this paper an *Artificial Intelligence* project is a project involving some form of logical inference by symbolic means. Other

[3] today, in Deductive Databases the terms *intensional* and *extensional* are used for expressing this distinction

important activities in AI like natural language processing, image processing, robotics, knowledge representation and extraction or computational research on models of the human mind will be disregarded.

Researchers in symbolic inferencing and theorem proving in the early development of AI, used first order predicate logic as a universal representation language. *Resolution* based theorem provers were used and obtained sometimes reasonable results. Resolution is a strategy for theorem proving which hardly resembles the strategies used by humans. For a full treatment of the strategy I refer to standard textbooks like [3, 14]. The strategy requires several steps. The logical expression of assumptions and the (negated) conclusion are transformed into a collection of *clauses* build from *literals* having *terms* as arguments. The resulting list of terms is subjected to an iteration of *unifications, substitutions* and *resolution* steps, aiming for a derivation of the *empty clause*. If this succeeds one obtains a *refutation proof* showing that the negated conclusion is inconsistent with the assumptions thus proving the required result. If the resolution proof *fails* the substitutions computed can be used for constructing a *countermodel* for the conclusion.

The bad news is that the resolution strategy is highly nondeterministic: *indefinite clauses* lead to a branching structure of cases which all should be explored. Moreover, iterated substitutions on terms occurring in the clauses may lead to a combinatorial explosion. Worst of all, the method is incomplete, as some satisfiable formulas turn out to posses only infinite models, which will never be searched out to completion in a resolution proof.

In many practical situations this combinatorial explosion doesn't occur, which makes actual work in AI feasible. People saw a reason why this happened. Frequently the formulas used belong to the fragment known by the name *Horn clauses*, clauses with at most one *positive* literal. This property reduces the number of available literals on which a clause may be resolved to at most one, and thus one of the important sources of the combinatorial explosion is removed. The nondeterminism of the resolution method can be replaced by the deterministic depth first search backtracking strategy known today by the name *SLD resolution*. The language *Prolog*, which originated out of a natural language processing project but which to a large extent was founded on Kowalski's work on resolution theorem proving, actually has SLD resolution for the Horn Clause fragment as its computational model. [6]

Using SLD resolution the combinatorial explosion due to the growth of terms remains. However if we remove from Prolog the function symbols in terms (together with many features residing outside the computational core like build-in predicates having side effects) we obtain the "pure" core language known today by the name *Datalog*.

One of the claimed advantages of the language is its *declarativeness*: a Prolog program is a list of first order formulas which express the structure of the model to be constructed, just written in a rather unconventional dialect of predicate logic. On the other hand there exists a Prolog interpretator which actually builds the model described by the rules, and the user doesn't have to be bothered about the details on how the interpreter obtains its results. Note however that there is a major semantic difference with the interpretation of logic used by ordinary mathematicians: in predicate logic a set of formulas describes a family of models, whereas the Prolog interpreter produces a preferred *minimal* model. For people working in AI this difference was no problem at all, since they were looking for minimal models as interpretations anyhow. Curiously enough such a minimal model is also taken to be the natural interpretation for the contents of a (relational) database.

A major contribution to the popularity of Prolog was the selection around 1982 by the Japanese of this language as vehicle for their Fifth Generation Project, and the impact this choice had on research in computer science around the world.

Logic programming is the Computer Science research area where the above intuitive notions are investigated and formalized. Notwithstanding the fact that the field dates from the 1970-ies it lasted until the early 1990-ies before a complete satisfactory mathematical correct formalization of the theory of logic programming which was free from the errors caused by the interaction between syntax and semantics, appeared in the literature.

1.2 The players

In the centre of the scene a group of people is having a picnick. They are discussing how they are related to each other...

We stick to the traditional example of family relations among humans to illustrate the similarities and differences between the various Computer Science fields mentioned above.

Consider the statement that Dorothy is an aunt of Jessica. This can describe two different relations: it may be the case that Dorothy is a sister of one of Jessica's parents, or it can mean that Dorothy has a husband Fred which is a brother of one of these parents, thus becoming Jessica's uncle. It can even mean both relations at the same time. Sisterhood is a defined relation as well. Dorothy is a sister of Thomas, since they share parents and are not the same person; moreover Dorothy is a female person. Parenthood and marriage are relations which are not defined in terms of other relations, and the same holds for the gender, date of birth and name of people. Existence of a set of entities called people is assumed, together with the possibility of identifying people by name. These notions are considered to be basic notions when talking about family relations. A further distinction is that gender, date of birth and name are *unary* properties of people, whereas parenthood and marriage are *binary relations*. In fact marriage has a hybrid nature: as a relation it connects two people, and at the same time it is an *entity* in itself having properties like a date of marriage (and possibly a date of termination).

First order logic is a convenient language to express such relations. In this particular case we might write something like:

$$parent(Gerti, Jessica) \wedge sister(Gerti, Dorothy) \Rightarrow aunt(Dorothy, Jessica)$$

or preferably, since it is a general rule, using logical variables

$$\forall G, J, D[parent(G, J) \wedge sister(G, D) \Rightarrow aunt(D, J)]$$

a similar rule for sisterhood reads:

$$\forall X, Y, Z[parent(Z, X) \wedge parent(Z, Y) \wedge X \neq Y \wedge female(X) \Rightarrow sister(X, Y)]$$

Information about the primitive relations in logic is expressed by atomic assertions like: $female(Gerti)$ or $parent(Gerti, Jessica)$.

The corresponding rules written in a Prolog become:

```
aunt(D,J) :- parent(G,J), sister(G,D)
sister(X,Y) :- parent(Z,X),parent(Z,Y),X ≠ Y, female(X)
female(X):- person(X,'F',Any) .
```

showing that the order of the implication is reversed, and that the quantifiers are omitted (by convention free variables are understood to be universally quantified). The basic information is presented by means of *facts* which are stored in the Prolog *database* which in fact is just a collection of lines in the program:

```
person(Gerti,'F',19240713).,
marriage(Fred,Dorothy,19411013).
```

which illustrates that the unary properties of entities are written in a format resembling database tuples, rather than using a collection of unary predicates.

In a Relational Database model people will be represented by a (stored) database table whose structure can be expressed by a structure declaration like:

```
PERSON(NAME:STRING(20),GENDER:{'M','F'},BIRTH:NUM(8) # KEY(NAME) )
```

Gerti now can be represented by a tuple like `(Gerti,F,19240713)` in the `PERSON` table. Other base relations describing parenthood and marriage. are represented by relations as well:

```
PARENT(PAR:STRING(20),CHILD:STRING(20) # KEY(PAR,CHILD) )
MARRIAGE(HUSB:STRING(20),WIFE:STRING(20),DATE:NUM(8)
# KEY(HUS,DATE), KEY(WIFE,DATE) )
```

and by inserting the appropriate tuples the basic family relations are described.

Having provided the required facts and rules we hope that our computer system indeed will be capable to derive the fact that Gerti is an aunt of Jessica.

In Prolog one may submit a query like: `?- aunt(Dorothy,Jessica)` hoping that the system will return the answer `'YES'`. The Prolog interpreter, using its SLD resolution, will first discover that Gerti is a parent of Jessica, and subsequently attempt to solve the subgoal of establishing that Dorothy and Gerti are sisters, which eventually will succeed by invoking the rule for sisterhood, but not before having inferred along the way that Dorothy is female.

Extracting the same information from the relational database would require to invent an algebraic or logical relational expression representing the aunt relation. We might describe the sister relation by:

```
SISTER(A,B) =
SEL( PROJ(( PARENT(X,A) JN PARENT(X,B) ,(A,B) ) , A ≠ B )
JN PROJ( SEL(PERSON(A,G,Arb), G = 'F', (A) )
```

and this complicated expression must be joined with the parent relation to obtain one of the two contributing relations for aunthood. Once the view definition for aunthood is constructed one can evaluate it on the database hoping that in the resulting table the tuple `(Dorothy,Jessica)` will show up. If the database contains a large number of people it might even be advantageous to submit a query selecting the tuple `(Dorothy,Jessica)` from the view describing aunthood: the database optimizer may push the selection down in the algebraic expressions thus reducing the size of the intermediate results and the processing time for the query.

Observe the distinction between the definition of this derived sister relation (called a *view definition*) and the table produced by evaluating it in the database (the *view instantiation*).

This example illustrates what declarativeness is all about: in Prolog we have specified the rules and the system navigates towards a solution. In a Relational database we have to do a lot of thinking before we can submit the right query to the database.

1.3 The Play

Disturbed by the fierce debate the engineers and logicians drop their activities and join the family, in an attempt to solve the dispute...

In the above example the Prolog approach offers a reasonable degree of declarativeness. However, inspection of how the Prolog inference process will proceed if run against a substantial fact database, will show that the computational process can be quite inefficient. Moreover, there is no actual database system against which the Prolog system operates; the Prolog system contains a fact database stored explicitly in the program. Consequently none of the secondary benefits of database technology like persistence, security and sharing will be available.

Declarativeness is also a property ascribed to the Database Interface language SQL. Since SQL is claimed to be *relationally complete* it should be possible to formulate an equivalent of the definitions given in Prolog. However, the resulting expressions will become far more complex than the ones given in the previous section. Worse, in the version of SQL available around 1982, due to the lack of a Union operator, it would have been impossible to express the aunt relation by means of an expression, and one would need a process for evaluating this relation.

Even if the database interface language is truly relationally complete the problem remains how to *obtain* the required relational expressions. In Prolog, as in mathematics and logic, a complex definition is expanded into a chain of more simple definitions, terminating at basic notions. The corresponding feature in relational databases would be a *view definition* being expanded into a chain of other view definitions, ultimately arriving at the tables stored directly in the databases. This requires view definitions to be invokable as easily as stored relations. Yet another feature which the database systems around 1982 didn't offer [30].

Finally the translation itself. Life will remain hard if every Prolog definition must be translated by hand into a corresponding relational expression. The translation, however, is not hard to automate. The strategy how to perform it can be found in textbooks like [28]. Yet there remains the difference between knowing how to do it in principle and actually building a system doing it. Such automated systems are now known by the name *Deductive Databases*.

The ideal of Deductive Databases is to combine the user friendly declarative world of Prolog with the information crunching power of existing relational database systems. Isolate a fragment of Prolog which can be evaluated by means of relational algebra on a database system, automate the translation, and build a system which uses as much computational power from the database engine as feasible. Next enjoy using for free the secondary benefits of the database system which you no longer have to reinvent for your AI system: sharing data, concurrency, transaction management, security and so on...

The core fragment of Prolog for which this translation exists is known by the name *Recursion-free Safe Datalog*. This fragment consists of rules describing predicates in Prolog defined by a non-recursive[4] system of Horn clauses, starting from a finite collection of basic relations. Terms do not contain function symbols. The rules moreover satisfy a *safety condition* preventing the occurrence of free variables in the answer as given by the Prolog interpreter. This condition is required in order to keep the derived relations bounded. For such rules the defined predicates have an extent which can also be described by algebraic relational expressions having base tables corresponding to the basic relations as atoms. The translation process moreover can be mechanized.

As illustrated by the extensive literature on the topic *Logic and Databases* around 1980 this semantic relation itself was reasonably understood. Yet most projects in this area worked on the basis of the so-called *tuple at a time* approach. The AI system no longer needs to store its fact database inside the program, if the resulting facts can be represented by tuples in a relational database. But the combinatorial processing of this information is still performed at the side of the AI inference engine and therefore the power of the database system itself is severely under-utilized. None of the secondary benefits of database technology becomes available to the AI system.

One may wonder why so few projects in the early 1980-ies went beyond this tuple at a time approach. As we conjecture, the lack of an appropriate relational database system may have been an important cause....

2 The Prolog - BS12 Interface project

Inspired by the hype provoked by the Japanese Fifth Generation project and its challenge to the rest of the world [22] the first author, using her experience as a participant for several years in the development of IBM's product Business System 12, noted the correspondence between backtracking in Prolog and evaluating expressions in a relational database. Initial ideas on this approach were presented at a workshop at the Free University in Amsterdam, early 1984 [17]. This workshop was dominated by proponents of the tuple at a time approach resulting is disbelief among the audience. Still convinced of the suitability of BS12 as a vehicle for realizing this idea, she arranged a student internship at IBM for C.J.F. Doedens, who developed a working prototype within 9 months. His report [12] constitutes the master thesis on which he graduated in Computer Science at the University of Amsterdam. See also [13]. The best accessible document describing the project is [20].

Since this paper focuses on the general ideas and historical backgrounds of the project we will not present a complete technical description of the translation process. The theory needed for constructing such a translator can be found in textbooks like Ullman [28]. Instead we discuss the requirements on an RDMS in order to be a suitable platform for realizing a compilation based deductive database system. BS12 indeed satisfies these requirements.

[4] The full language Datalog allows for recursive definitions also; in the last section of this paper we explain how such recursive rules can be treated in a compilation based deductive database.

2.1 Business System 12

Business System 12 is the name of a Relational database system developed by IBM, the Netherlands in the early 1980-ies, building on the work of a group at Peterlee, UK. During the early stages of development there was little interaction with database development projects going on in the USA like system R. Blauw and Duyvesteijn from the University at Twente served as advisors. A first version of BS12 became operational in 1983. The system was offered as a service from the International Network in Zoetermeer in the Netherlands. There was never an installation delivered to clients; users accessed the system in a time shared environment. IBM management at some point in the 1980-ies decided no longer to support the product, after which the system gradually disappeared. However, a major application running on the system was a financial project build around the XSHARE database storing stock exchange rates of some 100000 companies over a period of five years, resulting in the (for this era tremendous) volume of 5GB! The fact that it turned out to be impossible to migrate this application to DB2 without loss of performance and functionality was the main reason why BS12 actually survived until approx. 1996.

It is difficult to find in the literature information about BS12; the reference manual [2] contains a full description but was made available primarily to users. A more superficial description of the functionality in the style as advocated in [26] constitutes the unpublished document [16]. A paper on concurrency issues appears in the proceedings of a Dutch national conference [7]. Andrew Warden, in reality Hugh Darwen, one of the designers of BS12 mentions the system with hindsight in [30].

In BS12 tables are relations, subsets of Cartesian products of domains, where columns are identified by named attributes; the rows correspond to the tuples in the relation. Domains are finite sets relating to one of the five basic types: **Character, Numeric, Name, Bit, Timestamp**. The basic type **Name** represents a sort of strings with a normalization convention. It is the datatype used for table and attribute names. To every domain a default value can be assigned. Duplicate rows in a table are disallowed. Tables are homogeneous: all rows have the same attributes and domains. The ordering of rows in a table is insignificant.

The environment where the system had to operate: a time sharing environment where several customers are using the database system simultaneously, had severe consequences for the system architecture. Since competitors might share the use of the system they had to be protected against each other, both with respect to accessing information and the denial of service. Consequently, having a global catalogue of tables and/or views was unthinkable. Every user had access to his/her private dictionaries with the possibility of opening items for shared use.

A second important feature is the *monolingual* structure of the system interface. BS12 focusses on tables, being relations interpreted as sets. But processes, functions etc. are represented by objects which are also tables (*language tables*) which makes these objects accessible to relational operators. System information like the catalogues of tables, columns, or views are also stored in tables.[5]

A third important feature is the fully algebraic character of the system. The number 12 in the name BS12 refers to the presence of 12 relational operators:

[5] This makes it possible to construct the "paradoxical" Russell view consisting of those views which don't contain themselves as a row.

- `SELECT` set comprehension on a table based on a selection criterion (subset of rows)
- `PRESENT` a hybrid combining projection and renaming (subset of columns)
- `CALCULATE` extends a table by a new attribute which functionally depends on the others where the dependency is given by an explicit formula
- `GENERATE` production of a single row relation computed from constant values
- `SUMMARY` a aggregate construct used for grouping, summation etc.
- `JOIN` the natural join
- `MERGE` a version of the outer join where nonmatching row are extended by default values rather than nils
- `QUAD` Cartesian product of relations
- `DIFFERENCE` relative set difference
- `EXCLUSION` symmetric difference of two tables
- `INTERSECTION`
- `UNION`

The Boolean operations (the last four in the list) do not require that the two argument relations are union compatible; if they are not the resulting table is projected on the attributes shared by both operands.

BS12 is algebraically complete: wherever a base relation can serve as an operand in an expression its role may be performed by a relational expression as well. Note that the result of evaluating an expression in this context always represents a relation considered as a set. Every relational expression in the algebra represents a *view* on the database. This view can not only be evaluated but the expression defining the view can be stored in a language table, which itself can be added to a user owned catalogue called the Table of Views. Moreover these definitions can be stored and modified during a session.

As we will indicate in the sequence both the algebraic completeness and the monolinguality are instrumental in making BS12 a suitable platform for the construction of a deductive database system.

2.2 The translation of safe non-recursive Horn-clauses into view definitions

Given a system of Horn-clause rules for a family of predicates, and given a collection of basic predicates used in these rules we want to interpret this systems as a definition of a collection of predicates in terms of the given basic ones. Assuming that the basic predicates correspond to stored tables in a relational database we want to obtain for the derived relations relational expressions in terms of the basic tables with the intended meaning.

In a preprocessing stage the clauses are ordered such that clauses sharing a predicate in the head are grouped together. The group defining a predicate which occurs in a subgoal in the body of some clause should precede the group containing this clause. Consequently clauses all whose subgoals invoke predicates corresponding to base tables or build-in predicates appear at the beginning of the list. If the system of Horn clauses is not recursive such an ordering can be obtained by topological sorting. For recursive families of Horn clauses such an ordering doesn't exist.

One additional complication may lead to meaningless expressions. The complication involves the occurrence of variables in the head which don't appear in the body, or only appear in build-in predicates. Semantically this means that the values assigned to the corresponding attributes in the relational tabel for this predicate have no designated domain and thus give rise to potentially unbounded relations. The problem is illustrated by rules like: `mortal(Everybody).` or `larger(X,Y) :- (X > Y).`

The easy way out is to prohibit such unbounded relations by enforcing a so-called *safety condition* on Prolog programs.[6] The condition stipulates that in each rule all variables occurring in the head occur *bounded* in the body: Constants and variables occurring in a basic predicate or a predicate earlier in the order of definitions in the body occur bounded in the body. The same holds for variables occurring in a build-in predicate whose semantics enforces this variable to be functionally dependent on other bounded variables. All other variables are unbounded.

The translation of Horn-clauses into view definitions as described by Ullman [28] consists of three stages.

In the first stage a relational expression is constructed for every subgoal based on a relational predicate; the build-in subgoals are treated separately during the second stage. These relational expressions basically consist of the relation or view representing the predicate in the subgoal. However, due to the occurrence of constant arguments or multiple occurrences of a single variable the `SELECT` operator has to be invoked.

In the second stage the various expressions corresponding to the subgoals are joined together to yield an expression corresponding to the extent of the body of the rule. Build-in predicates (in this context primarily (in)equalities on numeric arguments) are translated to another select. Moreover, in order to accommodate for the positionally specified arguments in the subgoals in a relational algebra based on named attributes, frequently the expression corresponding to a subgoal frequently will have to be *renamed*. `RENAME` is also essential in order to arranged that shared variables between subgoals will correspond to common columns in a `JOIN` expression.

In the third and last stage the contributions of the various rules for a predicate must be put together to provide an expression which defines this predicate. A `UNION` of the subgoal expressions, after a `PROJECT` on the variable occurring in the head of the rule will suffice in case these subgoal expressions are union-compatible and have the same signature as the head predicate. However, due to the occurrence of repeated variables and constants in the head of individual rules the subgoal expressions in general are not union compatible. However, the semantic impact of such occurrence patterns in the head of a rule can also be expressed by extending the body with a few additional build-in subgoals. Ullman uses a preprocessing stage called *rectification of rules* which solves this problem. The impact is that the expressions for the body will include some more select conditions.

The translation constructs for every defined predicate a corresponding relational expression which either can be evaluated on the database or which can be stored as a view definition (or both), depending on what your database system has to offer. From the above description we moreover can infer the requirements on the functionality to be satisfied by the RDMS in order to support a Datalog compiler as intended.

1. The translation requires that the relational operators `SELECT, PROJECT, JOIN, RENAME` and `UNION` can be used in expressions in an arbitrary nested way. Consequently, with

[6] Our safety condition follows [20]; it is slightly more general than the one given by Ullman [28], who requires equality in stead of functional dependence.

a system like the earlier version of SQL which requires to form a Union by means of a process rather than an expression life will be quite unpleasant...

2. If systems of rules, rather than rules for a single predicate have to be translated conveniently, one needs to be able to invoke defined views on the same positions as basic tables. Views must be first class citizens.
3. In order to be able to add, remove and modify rules during a session the corresponding views must be inserted, deleted and modified in the database, without having to restructure the entire database after every change. This requires that the catalogue storing views and their definitions should be readily accessible and updatable during runtime.

As indicated before BS12 fully satisfied these three requirements. The `PRESENT` operator combines the functionality of a project and a renaming operator and the other three required operators: `SELECT, JOIN` and `UNION` are available. In the real world, the SQL based systems available in 1984 violated them all (a situation which, in the more recent SQL2 standard, is hardly improved since). This illustrates both why BS12 was the ideal RDMS for building our prototype and also explains why other groups never opted for a fully compilation based approach to deductive databases.

3 The aftermath; extensions of the functionality of the compilation based approach

Full Datalog includes a feature not covered by the translation process as described in the previous section: *Recursive definition of predicates*. Since Deductive Databases were rather popular in an academic environment where recursion is a well understood mechanism it was the first feature one wanted to add to the interface.

Full Prolog uses function symbols for building complex terms which are used for representing complex entities. These structures show a similarity with the structures in the nested relational model. A second extension involves the migration of our compiled based approach to a nested relational model.

Prolog itself supports the use of negated subgoals, by means of the *negation by failure* strategy. A much milder form of negation is offered in the relational database model by the `DIFFERENCE` operator. Therefore one wants to investigate whether some “tame” use of the negation operator can be added to the interface.

Finally, having numerical values in a Database is hardly useful if you can't compute on these values, preferably by invocation of symbolic expressions. One would like to see the functionality offered by spreadsheet programs to appear in a database system where the numeric constraints are described symbolically in the database scheme. This extension of functionality is investigated in the field called *Constraint Logic Programming* today.

Between 1984 and 1997 we have been involved with research projects aimed at the above extensions. Recursion and (nonrecursive) terms has been processed on our prototype system. For the negation the solution which is described in theory [28] evidently could have been implemented on our prototype, but we never did so. Finally, the extension with constraints became the focal point of the *RL-project*, which was initiated at the IBM San Jose Research Center[7] in 1985 and which was continued at the University of Amsterdam.

[7] now known as the IBM Almaden research center

In this section we will discuss each of these extensions.

3.1 Beyond Relational Algebra: Recursion

Recursive definitions of predicates in Prolog are as little a part of logic as recursive definitions are allowed in the branch of mathematics called Algebra. They violate the basic principle in mathematics that defining an object in terms of itself is disallowed. Yet logicians, computer scientists and Mathematicians have learned to understand and operate on the notions defined using recursion. These "definitions" better be understood as equations, and equations may have a solution. In fact, in order to be meaningful these equations better have a unique, or at least a *preferred* solution. As it turns out this frequently happens to be the case (at least in those situations where indeed recursive definitions nowadays are commonly used, since otherwise...).

The existence of a preferred solution in computer science often can be based on the semantic principle attributed to Knaster and Tarski, known as the *least fixed point principle*. If interpreted over some abstract mathematical structure Ω a recursive definition can be interpreted by means of an equation of the form

$$X = \Phi(X)$$

where Φ denotes some *functional* over Ω, a mapping from Ω into itself. Assume that Ω is a structure on which some partial order $\sqsubseteq$ is defined, such that there exists a unique minimal element $\perp \in \Omega$, and such that least upper bounds for countable increasing chains are defined: if $X_0 \sqsubseteq X_1 \sqsubseteq X_2 \ldots$ is a countable increasing chain then $\bigsqcup_{n=0}^{\infty} X_n$ denotes this least upper bound.

The functional Φ is called *monotonous* if $X \sqsubseteq Y$ implies that $\Phi(X) \sqsubseteq \Phi(Y)$. The functional Φ is called *continuous* if it satisfies $\Phi(\bigsqcup_{n=0}^{\infty} X_n) = \bigsqcup_{n=0}^{\infty} \Phi(X_n)$.

The Knaster Tarski principle states that for a monotonous continuous functional Φ the equation $X = \Phi(X)$ has a minimal fixed point, which moreover can be computed by $\bigsqcup_{n=0}^{\infty} \Phi^{(n)}(\perp)$, where $\Phi^{(n)}(\perp)$ denotes n-fold iteration of Φ on $\perp$.

This principle has been used to assign meaning to recursive procedures in programming theory with great success [1], where it forms the theoretical foundation of the *Scott induction rule*.

For the semantics of Prolog rules, and its interpretation in the relational database model, we take for Ω the family of relations within a (fixed) Cartesian product of domains, where $\sqsubseteq$ denotes set-inclusion and $\perp$ denotes the empty relation. It is easy to see that the five relational operators invoked in our translation process: `SELECT, PROJECT, JOIN, RENAME` and `UNION`, all are monotonous and continuous, and so are the complex expressions produced by our translator.

In applying this principle one has to be aware that in general the use of recursion in Prolog invokes a set of predicate symbols rather than a single recursive predicate. After translation this would define a collection of relations in terms of themselves. However, by forming the Cartesian product of these relations, the system can be replaced by a single relation having a recursive definition, after which the Tarski Knaster principle can be applied.

Hence we know what the meaning of a recursively defined relation should be. Moreover, the Knaster Tarski principle tells us how to compute this meaning. Starting with the empty relation we repeatingly submit the result obtained so far to the functional until no new tuples appear in the result. Where in the theoretical model this might require an infinite number of steps, in the database the process is guaranteed to terminate after finitely many steps, due to the fact that after all the domain Ω is finite. Thus increasing chains must become stable at some point, and it is easy to see that the first chain element X_i satisfying $X_i = \Phi(X_i)$ indeed equals the minimal fixed point.

From a practical perspective this evaluation method (the so called *naive evaluation scheme*) is highly inefficient since it will recompute the same tuples over and over again. The *semi-naive evaluation scheme* is an attempt to prevent some of these recalculations by rewriting the functional in such a way that only the "incremental part" is computed.

Much effort has been invested in building interfaces on databases capable of dealing with recursive definitions. In our project [20] we have performed some experiments with recursive rules using a compilation based approach. Rather than resubmitting the intermediate results as a table to the functional we compile a series of increasingly complex expressions representing the result of k-fold application of the rule and compiling it in relational algebra. This approach was enabled due to the architecture of BS12 where the user has access to his view definitions in the form of language tables. Furthermore the optimizer in BS12, which operated on some tree representation of the algebraic expressions, worked in such a dynamic fashion that the recursive expansion of the expressions and the optimization worked together in an incremental fashion. This made it possible to optimize these expressions at construction time. Once more the architecture of BS12 enabled us to investigate an approach followed by few others...

3.2 Beyond Datalog: Terms

Where Datalog supports as arguments in predicates only constants and variables, full Prolog allows terms build by application of functions to a number of terms also. Semantically a term violates the first normal form condition in databases where all attribute values have to be atomic values.

A typical example of a pair of Prolog facts illustrating terms is given by: `owns(Lucy,Mug).` in combination with `owns(Lucy,book(Homer,Ilias)).`

These two facts show that Lucy owns a mug for which no structure information is available, and a book with given author and title. In a Database property of the first sort is easy to represent; just store a tuple `(Lucy,Mug)` in the `OWNS` table. Property of the second type requires that one introduces another table `BOOK` for storing information about books, and subsequently storing in the `OWNS` table a tuple `(Lucy,X)`, where `X` is something like a *foreign key* value referring to the `BOOK` table. At a slightly higher level of abstraction one would consider this value `X` to be an *object identifier*. But then we are leaving the relational model proceeding towards something like a nested relational or fully object oriented model.

This example illustrates a rather evident strategy for bringing Prolog terms within the scope of our compiled approach. However, several problems remains to be solved:

- Storing a foreign key or an object identifier requires some mechanism for generating these values, and since in the Prolog world these values are not required it becomes the task of the database system to do so. It is questionable whether the database will be able to figure out whether it is attempting to create a new object identifier for a tuple which it has seen already before: duplicate removal becomes a far more serious issue, like it is in the world of Object Orientation.

– Inventing a domain containing the entities which can become property is non-trivial since property can both be unstructured and structured. In Prolog this is no problem since Prolog is an untyped language.

In 1986 S.J.C. Elbers, a student in Mathematics at the University of Amsterdam, was given the assignment for his master thesis to construct at IBM Uithoorn an extension of the prototype of our Prolog-BS12 interface which would incorporate terms [15]. We knew that the first problem would be easy to solve given the fact that BS12 in its internal structure generates object identifiers (called *pseudokeys*) for all tuples anyhow. It was just a matter of making these pseudokeys available to the user of the system. The developers at IBM at that time turned out to be willing to design the required patch on its system for the purpose of our project (keeping in mind that it might be of use for themselves as well. . . .)

The second problem was solved by systematically replacing columns by a collection of three columns, thus supporting the choice between an atomic value or an object identifier for all attributes. For atomic values the first column stores the value and the remaining two obtain a default value. For object identifiers the first column is a default value and the second and third column contain a table reference and a pseudokey value for that table respectively.

With this expansion of tables it turned out to be possible to extend our compiler in such a way that nonrecursive terms could be dealt with resulting in queries which provided the right answer. However we stumbled on yet another problem, this time having to do with Prolog itself.

Prolog, in its syntax hardly discriminates between functions and predicates. In the above example we might add a fact expressed by `book(Vergil,Aeneas).` This would be stored as an entry in the `BOOK` table as well. But a subsequent query for the existing books: `?-book(X,Y)` would now return the result consisting of the Aeneas only. Due to the fact that the Ilias is mentioned only as a term unification with the predicate would fail for the Ilias. Evidently in our database table storing books this distinction would be lost.

In order to preserve this (counterintuitive?) behavior of the Prolog system Elbers had to divide each relation in the database into two sections, one corresponding to the predicate information and another representing the so-called *argument information*.

The resulting system became rather complicated, but it worked. Proving its correctness became however hard. Elbers has argued for its correctness by modeling his database design into Prolog yielding a *Flat Prolog* version of a given program. Next one could show that the Flat Prolog version of the program returns always the same answers as the original one. Subsequently, showing that the compiler preserves the meaning of the flat prolog program was not hard.

In this extension recursive terms were disallowed; they could give rise to cyclic structures which would force the database system into an infinite loop if traversed.

3.3 Adding Negation: Stratified Programs

Negation by failure is based on the principle that a positive fact is false if it can't be shown to be true. Its correctness is tightly connected to the minimal model interpretation shared by Logic Programming, Artificial Intelligence and Databases. Yet, in a system whose proof calculus is incomplete there remains a difference between something being not true, and failing to prove something. Infinite (counter)models once more may cause big trouble.

Keeping in mind a database model, there shouldn't be to many problems. After all the Universe is finite, and therefore infinite models and infinite computations constructing them shouldn't arise at all.

Indeed, for pure recursion free Datalog the extension with negation is simple. There is a minor conceptual problem with the correct interpretation of variables which occur in negated subgoals only. For example, in a rule like:

```
bachelor(X) :- male(X), ¬married(X,Y)
```

the reading where all variables are implicitly understood to be universally quantified turns out to be incorrect, and not to be in agreement with what the Prolog interpreter yields. Achab who is happily married to Emily shouldn't be called a bachelor simply because he is not married to Katty also.

It is easy to computer the bachelor relation on a database. First on forms a table of all male persons. Subsequently one forms the projection of the marriage table on the husband attribute, and this projection is subtracted from the males:

```
BACHELOR(X) = ( MALE(X) DIFF PROJ( MARRIAGE(X,Y) , X ) )
```

Ullman [28] describes some strategies for translating negated subgoals into relational expressions; the main emphasis is to ensure that before a `DIFF` operator is used the two arguments should be made union compatible. The presence of negated subgoals also affects the formulation of the safety condition: occurrences of variables in negated predicate subgoals are not bounded occurrences: stipulating that some entity E doesn't have some property doesn't yield an indication on what the type of entity E should be.

What makes negation really troublesome is its interaction with recursion. The context of the right hand operand of the `DIFF` operator is not monotonous but anti-monotonous. This eliminates the justification for the Knaster Tarski minimal fixed point result on which the semantics of recursively defined predicates is based.

Indeed it is easy to present rules without a conceivable meaning by using negation and recursion in combination: well known paradoxes like the *Liar's paradox* in logic and the *Russel paradox* in set theory arise out of the interplay between self reference (recursion) and negation.[8]

In the model one may observe that the sequence of relations computed by iterating the functional on the empty relation will fail to be an increasing chain. It may grow in the first step, shrink in the second and so on. The sequence of iterates has no limit, but (given a finite Universe) end up in a no-trivial loop, each of which elements could claim to be the "correct" meaning of the recursive predicate.

The problem is solved by yet another syntactic restriction, which prohibits recursion in the scope of a negation. Predicates are grouped in *strata*. Stratum 0 consists of nonrecursive predicates and predicates defined by negation free recursion. Predicates of stratum $k + 1$

[8] In fact the Russel table mentioned in a footnote in section 2 is an example of this mechanism

are defined by recursive schemes, where all subgoal predicates occur in strata $0, 1, \ldots, k+1$, and where subgoals in stratum $k+1$ can't be negated.

For a stratified program a preferred semantics is obtained by evaluating the predicates in order of their strata: predicates of stratum 0 obtain their meaning just as before. Predicates in stratum $k+1$ obtain their meaning by the strategy from section 3.1 considering the meanings of predicates in earlier strata to be accessible as base tables.

By computing a series of (semi-)naive evaluations the semantics of stratified recursive programs, in principle can be evaluated on a database system. Stratified programs is about the largest class of programs for which a clear preferred semantics exists (there still is an extension of the notion, called *locally stratified programs*, where the tuples rather than predicates are stratified in layers, but that property is in general not syntactically testable).

In our paper [20] we observe the possibility of dealing with negation but we never bothered about actually implementing it in our prototype. The theory is excellently explained by Ullman [28].

3.4 Constraints and Elimination: the RL-project

Having numerical attributes in a database creates the wish to compute on such numerical values. Standard databases provide a restricted support for doing so.

On the one hand there are simple *calculate* operations where a new numeric attribute value is added to each tuple in a relation according to an explicitly specified formula (as implemented by the `CALCULATE` operator in BS12. The calculate operator doesn't belong to the core of relational algebra as required for relational completeness. It is unclear whether relations extended by a calculate can be subjected to further algebraic operations. For example: is it permitted for a user to perform a select on a attribute value which is computed to be the sum of two others?

On the other hand one has so-called *aggregate operators* where a table is grouped on the base of one attribute values and for each resulting group some other attribute is summed up to a total. The collection of aggregation operators which may be invoked is fixed and depends on the database system selected: *summation, maximum, minimum, counting* and sometimes logic operations like *exists* and *forall*. These operators don't belong to relational algebra and are in general not expressible in first order logic or one of its database oriented dialects. The main reason for aggregates to exist in systems is that they are needed in practical applications like generating reports for companies. Needless to state that in most systems the user is not able to define his own aggregate operators (Stonebraker's Illustra being a notable exception).

As far as support for computations is offered it lacks one feature which is essential for databases: *reversibility*. When designing a database scheme one doesn't yet know what sort of queries will be submitted to the database. The selection which attributes will be submitted as input values in a query and which attributes will be asked for output is made when the query is submitted, and at that time the system will have to invent a navigation strategy through the available tables in order to generate the answers. Knowing in advance the queries which are likely to be submitted the database engineer may assist the system by creating the right indexes etc.

However, a calculate instruction is always understood to be evaluated in the direction given by the definition. Most systems lack the intelligence needed to infer that given attributes `A,B,C`, where `C` is calculated by `C = A + B`, the output value of `B` can be obtained from the input values of `A,C` by rewriting the calculate into `B = C - A`.

A more complicated situation involves systems: given the calculate instructions `C = A + B` and `D = A - B`, our knowledge on elementary mathematics learns us that we can compute the values of `A,B` from those of `C,D`, but we need a symbolic equation solver in order that some system can discover this fact. Since both equations involve two unknowns and a single known quantity both equations in isolation are underdetermined, and therefore reversal by a simple rewrite is not possible.

In Prolog the situation is about as bad. The build-in predicate `sum(A,B,C)` only can be invoked to obtain a binding for `C` once bindings for `A` and `B` have been given. Symbolic equation solving is out of question. Nowadays there exist extensions of logic programming which provide this additional functionality (the field of *Constraint Logic Programming*), but around 1986 not much work had been done. Moreover, the research done aimed at extending Logic Programming with capabilities to work with constraints, and not to bring constraints within the scope of database technology.

It would be useful in a declarative framework to have full reversibility available on symbolic numeric attributes. Consider applications describing a physical system like an electronic circuit, a business application computing the impacts of various tax regulations, or a physiological model describing the blood circulation in a human body. In such models there are far more quantities in the model than actually will be stored in a table of data. Some quantities are fixed *constants* which preferably are written symbolically, some indeed are *observable* and *known*, some are *unknown* but *wanted*, regardless whether they are observable or not and finally, there may be values which are unobservable, unknown and unwanted which simply arise as *intermediate* values in the equational model.

Full declarative reversibility means in this context that one describes the model by giving the system of equations, and submitting the information about the partition in known, wanted and intermediate variables at a later stage. The system must invent an evaluation strategy to obtain the wanted variables from the known ones, eliminating the intermediate ones during the process. If a set of experiments is submitted rather than a single one the construction of the strategic evaluation plan should be performed just once, after which the execution of the plan on all input tuples should proceed with all the speed the database engine can provide.

In 1985 the second author held a visiting scientist position for eight months at IBM San Jose, as a member of a research group on Office Automation chaired by Peter Lucas. The aim was to establish a repository for business rules, to be accessed by multiple application programs, similar to the way a Database stores raw data. Business rules were to be understood to involve primarily algebraic numeric constraints (Austrian Social Security regulations being a prototype example). It became the author's task to invent a language for expressing contents of the repository.

The resulting proposal for a language called **RL** appeared in [18]. This report presents a syntax for rules, and a semantic model for their meaning. The model was highly influenced by our experience from the Prolog/BS12 interface.

Rules could be given in three formats: *Tabular rules* correspond to relational database expressions (view definitions); *Clauses* correspond to Horn-clauses in Datalog and *Constraints* represent the algebraic constraints on symbolic attributes.

Both Syntax and Semantics are based on the principle *Everything is a relation but not all relations are equal*. For example both tabulars and clauses describe named relations whereas the constraints describe an unnamed "world" relation. Tabulars are given as a program-wide global definition, whereas new clauses for an existing predicate can be given. Adding a clause behaves semantically as a union, whereas adding a constraint behaves as an intersection.

In order to support nesting for the three definitional mechanisms the language **RL** consists of *modules*. Evaluating a module amounts to first evaluating its tabulars, and next its clauses. Subsequently its constraints were modeled by a single relation on all attributes occurring in some constraint, which resulted from forming the intersection of the relations determined by the individual constraints (a perspective on the meaning of equations well known in the branch of mathematics called *Algebraic Geometry*). Subsequent modules could invoke the world-relation of previously defined modules as a tabular named by the name of the module.

The language as proposed included also mechanisms for user defined aggregates and abstract data types. The reception of the proposal was negative. Implementation of the ideas was expected to require at least three years. No activities in that direction took place at IBM. In fact the Office Automation project was killed several months after we left IBM.

The RL project was revived at the University of Amsterdam. For his master thesis Computer Science at the University of Amsterdam S.J. van Denneheuvel had written a symbolic equation solver in Prolog, to be used in some entirely unrelated project. Learning about **RL** he became interested in actually implementing it, and his equation solver turned out to be exactly the ingredient needed to do so.

If we restrict ourselves to the core functionality which an implementation should provide we are facing the following problem: Consider a collection of algebraic numeric (in)equalities over a finite collection Var of real valued attributes. This set Var can be decomposed into three subsets: $Var = K \cup W \cup I$, where K denotes the set of *known* attributes, i.e., those attributes whose value actually are stored in tables in the relational database; W denotes the set of *wanted* variables, i.e., the attributes whose values should be output in the answer relation. Finally I denotes the set of *intermediate* variables, being those who occur in the equations but whose value is irrelevant to the user and not stored in the database. The task is to figure out for which tuple of known variables present in the database the system of equations is solvable, and to compute for each solvable tuple the desired tuple of wanted variables in the solution.

To solve this problem we invoke the symbolic equation solver in order to perform the following two tasks. In the first place the intermediate variables should be eliminated from the system. This is done by isolating such a variable x in a single equation and rewriting the equation in an equivalent equation of the shape $x = f(\ldots)$. Next we substitute the right hand side expression for x in all remaining equations, and delete this equation. The impact of this step is that the number of intermediate variables has been reduced by one (and so has the number of equations). Once the intermediate variables are gone similar steps are performed for the wanted variables, except that the rewritten equations are preserved. If everything works out the resulting transformed system consists of a list of equations of the form $x = f(\ldots)$ for each wanted variable x, where the right hand side expression contains constants and known variables only: the *solution set*. The Remaining equations, containing known variables only, form the *residual set* of equations.

In this form the system can be translated into a select-join-project-calculate expression over the database: select those tuples of known variables from the join of the relevant tables which satisfy the residual system, extend them by calculates according to the definitions contained in the solution set, and project the result on the wanted attributes. So after preprocessing all answers are produced by a single database query. Once more BS12 would have been an ideal system to serve as a database back end, but by the time the project arrived at this point IBM in the Netherlands had lost all interest in experimenting with this system (the official policy at this time was to retire BS12 as soon as possible...)

Evidently not all problems have a solution. If the system is under-determined at best a symbolic solution can be provided. By returning the residual set and solution set in a symbolic format this answer can be presented. If the problem is overdetermined and inconsistent a diagnostic can be returned. If the algebraic complexity grows beyond the capacity of the symbolic equation solver and its elimination strategy fails the user will be informed.

The prototype build by van Denneheuvel formed the basis of his ph.d. thesis at the University of Amsterdam [9]. Papers describing the language, its implementation and an example of its use in a medical model are [10, 11, 23, 24].

After completing his thesis van Denneheuvel build a front end to his system which compiled **RL/1** to SQL. This tool turned out to be useful even in the case no constraints were actually used, since the **RL/1** code for a database application is far more compact than its SQL equivalent. The language was used as a rapid prototyping tool in the early stage of a major development project of Syllogic, a high tech software corporation in the Netherlands[9]. When it became clear how horrible the performance of the generated code was this approach was dropped. In order to be of real use the entire system will have to be redeveloped from scratch, and this remains to be done.

The last **RL** related project involves an Object Oriented extension called **OORL** which has been investigated in the years 1994–1997 by dr. E. Rotterdam as a postdoc at the University of Amsterdam. The report [25] describing this language which offers a declarative window on the inherently imperative world of Objects has not yet been finalized.

Conclusion

The projects we were involved with in the period 1983 until 1997 have proven that a compilation based approach to deductive databases is, in theory, possible. We never got to the point to determine whether the approach would yield acceptable performance for real life applications; this would require far more investments than we were able and motivated to spend. We also discovered that, starting from the clean mathematical model for the intended semantics, functionality from areas beyond datalog could be added, preserving a reasonable amount of transparency of the languages used.

These projects were made possible by circumstances which allowed to disregard the pressure to follow the main stream approaches investigated elsewhere. Moreover we found

[9] now a wholly owned subsidiary of Perot Systems

in BS12, a system which had been developed under a similar climate of independence, an ideal tool to perform experiments with our ideas.

References

1. de Bakker, J.W., *Mathematical theory of program correctness*, Prentice Hall, 1980.
2. Business System 12 Reference Manual, pre-release copy, IBM DCS-SC, Uithoorn, June 1983.
3. Chang, C-L., Lee, R. C-T., *Symbolic logic and mechanical theorem proving*, Academic Press, 1973.
4. Chen, P.P., *The entity-relationship model: toward a unified view of data*, ACM Trans. on Database Systems, 1(1), 1976, 9–36.
5. Codd, E.F., *Extending the data base relational model to capture more meaning*, ACM Trans. on Database Systems, 4(4), 1979, 397–434.
6. Colmerauer, A., Roussel, P., *The birth of Prolog*, in Bergin, T.J. jr., Gibson, R.G. jr., eds., History of Programming Languages - II, ACM Press, 1996,pp. 331–367.
7. du Croix, A.J., *Concurrency Considerations of Business System 12*, proc NGI-SION 1985 Informatica Symposium, NGI, Amsterdam, April 1985, pp. 501–509.
8. Date, C.J., *An introduction to Database Systems*, sixth edition, Addison Wesley, 1995.
9. van Denneheuvel, S., *Constraint solving on database systems; Design and implementation of the rule language RL/1*, ph.d. thesis, univ. of Amsterdam, Oct 29 1991.
10. van Denneheuvel, S., van Emde Boas, P., *The rule language RL/1*, in Tjoa, A.M., Wagner, R., eds., Database and Expert system applications, Proc. DEXA 1, Vienna, Aug, 1990, Springer Wien, 1990, 381–387.
11. van Denneheuvel, S., Kwast, K.L., van Emde Boas, P., de Geus, F, Rotterdam, E., *Symbolic computation in RL/1*, proc. DEXA'91, Berlin Aug. 1991, Springer Wien,1991, 559–564.
12. Doedens, C.F.J., *Logic Programming and Business System 12*, rep. TR 13.198 IBM INS-DC Uithoorn, Dec 1984; this document constitutes also the author's Master thesis in Computer Science at the Univ. of Amsterdam.
13. Doedens, C.F.J., *Logic Programming with a relational data base management system*, proc NGI-SION 1985 Informatica Symposium, NGI, Amsterdam, April 1985, pp. 485–500.
14. Doets, K., *From Logic to Logic Programming*, Foundations of Computing Series, the MIT press, 1994.
15. Elbers, S.J.C., *De compilatie van willekeurige Prolog structuren in een relationele database*, rep. IBM INS-DC TR 13.206, Jun 1986; (in Dutch); this document constitutes also the author's Master thesis in Mathematics at the Univ. of Amsterdam (rep. FVI-UvA-86-14)
16. van Emde Boas, H. , *Feature Analysis of Business System 12*, INS-DC IBM Uithoorn; unpublished internal memorandum, Oct 04 1984. (Facsimile edition: rep. ILLC-X-1998-01).
17. H. van Emde Boas-Lubsen, P. van Emde Boas *Logic programming in a relational database query language (working document)* in M.L. Kersten ed., Proc. Workshop Coupling PROLOG with Database management systems, VUA Jan 1984 Amsterdam, pp. 11–17
18. van Emde Boas, P. *RL, a language for enhanced rule bases database processing*, working document, rep. IBM research, RJ 4869 (51299), 1986
19. van Emde Boas, P., *A semantical model for the integration and modularization of rules*, proc. MFCS 12, Bratislava, Springer LNCS 233, 1986, 78–92.
20. van Emde Boas, H. , van Emde Boas, P., *Storing and evaluating Horn-clause rules in a relational database*, IBM J. of Research and Development, 30(1), 1986, 80–92
21. van Emde Boas, H. , van Emde Boas, P., Doedens, C.F.J., *Extending a relational database with logic programming facilities*, Rep. IBM, INS-DC Tr 13.195, Oct 1984
22. Feigenbaum, E.A., McCorduck, P., , *The Fifth Generation; artificial Intelligence and Japan's Computer Challenge to the World*, Addison Wesley, 1983.
23. de Geus, F., Rotterdam, E., van Denneheuvel, S., van Emde Boas, P., *Physiological modelling using RL*, in Stefanelli, M., Hasman, A., Fieschi, M., Talmon, J., eds., proc. AIME'91, Maastricht, Springer Lect. notes in Medical Informatics 44, 1991, 198–210.
24. de Geus, F., Rotterdam, E., *Decision Support in Anaesthesia*, ph.d. thesis Univ. of Groningen, Dec 02 1992.
25. Rotterdam, E., *OORL, a relational and Object Oriented Database Language*, in preparation
26. Schmidt, J.W. , Brodie, M.L., eds., *Relational Database Systems; analysis and comparison*, Springer 1983

27. Stonebraker, M., *Object-Relational DBMSs; the next great wave*, Morgan Kaufmann, 1996.
28. Ullman, J.D., *Principles of Database and Knowledge-base systems*, vol 1, Computer Science Press, 1988.
29. Ullman, J.D., *Principles of Database and Knowledge-base systems*, vol 2, Computer Science Press, 1989.
30. Warden A., *Adventures in Relationland*, in Date C.J., Relational database writings 1985-1989, Addison Wesley 1990, pp. 453–521.