Data Constructors: On the Integration of Rules and Relations

Matthias Jarke[1], Volker Linnemann, Joachim W. Schmidt

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-6000 Frankfurt/Main 11, West Germany

Abstract:

Although the goals and means of rule-based and data-based systems are too different to be fully integrated at the present time, it seems appropriate to investigate a closer integration of language constructs and a better cooperation of execution models for both kinds of approaches.

In this paper we propose a new language construct called **constructor** that allows the definition of new relations from existing ones by means of recursion. The constructor is semantically defined by the least fixed point of a set expression and blends well both with a strongly typed modular programming language and with a relational calculus query formalism. Moreover, it is shown to provide expressive power at least equivalent to the declarative semantics of PROLOG while avoiding some disadvantages of it, for example, poor modularity and infinite loops. Furthermore, the constructor is set-oriented thus allowing more efficient implementation techniques than those available through proof-theoretic methods typical of a rule-based approach.

1. Introduction

Combining the semantic capabilities of rule-based knowledge representation and reasoning systems with the efficiency-oriented mechanisms for query result construction and transaction processing in large shared DBMS has been the focus of much recent research [Kers 84]. Apart from the possibility of defining a completely new architecture for "knowledge base management systems", the solutions proposed so far can be interpreted as extreme points in a continuum of coupling strategies.

Researchers propose either to replace one system completely by the other [ScWa 84] - the end points of the spectrum - or to couple current expert systems languages (most notably, PROLOG [JaVa 84], [Zani 84]) with existing DBMS interfaces - the cutting point defined by history.

We believe, along with a number of other researchers [Smith 84], [Ullm 84], that a coupling strategy is preferable to fully integrated solutions. Because of the different stress on representation vs. efficiency between Knowledge Representation (KR) and Database (DB) research [MyBr 85], little is gained (and unnecessary complexity is incurred) by putting all capabilities into one system. In contrast, a coupling architecture allows each subsystem to evolve independently and to offload the reconciliation task to separate coupling tools [Jark 84].

Given that coupling is necessary, the question remains what the capabilities of each of the partners should be. While in the short run there is a clear economic incentive to leave existing systems as they are [JaVa 84], nothing indicates that the optimal division of labour between the deductive capabilities of rule-based systems and the selective power of data-based systems will remain where it has been historically - at the point of 'relational completeness' as defined by [Codd 72].

The question of exactly what capabilities should be added to the DBMS is open. [Ullm 84] proposes an architecture using 'capture rules' that define useful extended DBMS capabilities. The present paper follows a similar approach but investigates the problem from the viewpoint of integrated database programming languages rather than from a PROLOG perspective. Since database programming languages handle first-order queries as well as functions, iteration and recursion, they provide a framework powerful enough to implement any first-order derivation based on sets of facts. Current query optimization strategies, however, do not 'ake advantage of the relationships among the :orresponding sequence of queries. Efficiency)ecomes the responsibility of the programmer.

[1]Work done at the Graduate School of Business Administration, New York University

Rather than enhancing a query optimizer directly in order to handle multiple related queries, this paper studies special-purpose language constructs that capture higher-level data definition and operation and are easily recognizable by a compiler. To provide the necessary framework, section 2 reviews the database programming language DBPL which integrates relational data structures and transactions with the programming language MODULA-2 [Wirth 83].

The main focus of the paper is the detailed analysis of a DBPL extension called **constructor** which has evolved from the **selector** concept introduced in [MaReSc 84]. While selectors allow the definition of selected subrelation variables, constructors expand existing relations. Constructors implement recursion using an equational fixed point semantic. We introduce constructors in sections 2 and 3, and show how to integrate the tuple relational calculus concepts of negation and universal quantification into this framework. Moreover, we demonstrate that our proposal provides expressive power at least equivalent to PROLOG's clause-order independent declarative semantics yet remains faithful to the spirit of typed, procedural database programming languages, such as Pascal/R [Schm 77] or Adaplex [Smith 81]. In addition, our proposal eliminates PROLOG's problem of infinite loops which arises because of the implementation via backtracking [Covi 85].

The database programming language environment also inspires particular implementation and optimization strategies since it is frequently used for implementing higher level database interfaces. In section 4 we interpret constructed relations as an extension to range-nested expressions [JaKo 83], and outline a three-level compilation and optimization framework.

2. Types, Relations, and Predicates

The impact of logic on computing - from early data processing in the fifties to modern computer science - can hardly be overestimated.

In the field of programming logic marks the step from machine-oriented coding to algorithmic programming. High level languages provide conditional statements and boolean expressions, use propositions for data type definition, and depend crucially on predicates for the specification of language semantics and for reasoning about programs [Gries 81], [Hehn 84].

In the area of data modelling, the degree to which predicates are utilized allows a distinction between early reference-oriented data models and those that capture more of the relationships defined by the application semantics.

2.1. Data Types and Predicates

If "a type is a precise characterization of structural and behavioural properties which a collection of entities (actual or potential) all share ..." [Deut 81], the formalism by which those properties can be characterized decides upon the power of a type calculus.

Currently prevalent procedural programming languages only allow type definitions based on restricted propositional logic. Take, for example, the following Ada subtype definition:

```
partidtype IS RANGE  1..100,
```

which is equivalent to the domain predicate (1≤p AND p≤100) and defines the domain set

```
partidtype
   { EACH p IN integer: 1≤p AND p≤100 } .
```

The expressiveness of the type calculus in high level languages corresponds closely with that of the expression and statement part of these languages. As a consequence any action to be taken to assure type properties can be expressed directly in the language. A type checker can produce run time code in the source language to assure, for example, type correctness of an integer expression, ix, which is to be assigned to a variable, p, of partidtype:

```
IF (1≤ix) AND (ix≤100)
THEN p:=ix
ELSE <exception> .
```

Programmers reduce the possibility of run time exceptions by acquiring sufficient information on rhs-expressions through inductive reasoning about assignment chains and subtype definitions (and so do clever compilers).

Approaches to programming that are more concerned about correctness allow for the definition of additional program properties by so-called annotations. Ada annotations, for example, can be specified in the metalanguage ANNA [Krie 84], and Ada programs can be proven formally correct with respect to their specification. The meta language ANNA allows full first-order assertions, while the object language Ada is restricted to propositional logic. An Ada subtype definition, for example, primetype, can be fully specified by the following ANNA annotation [Krie 84]:

```
primetype IS integer ||
     WHERE p IN primetype ==>
        ALL n IN integer
           ((1<n AND n<p) ==> p MOD n ≠ 0),
```

defining the domain set

```
primetype
   { EACH p IN integer:
        ALL n IN integer
          ((1<n AND n<p) ==> p MOD n ≠ 0)} .
```

2.2. Predicates in Database Languages

Database models such as the relational model are very concerned about data integrity; they go beyond programming languages, therefore, in the sense that they provide the expressiveness of first-order logic directly through relational languages.

On the expression level, the request for "relational completeness" [Codd 72] of query languages is essentially met by allowing full first-order predicates, p(r,...), as selection predicates in relational expressions:

```
  reltype {EACH r IN rel: p(r,...)}.
```

On the type or schema level, the role of predicates can be exemplified best by comparing a Pascal-like set-type definition

```
  settype = SET OF elementtype,
```

with a relation-type definition.

The legal values of a relation are also sets of elements; they have to meet, however, the additional constraint that some attribute (or a collection of attributes) serves as a key, i.e., has a unique value amongst all the elements of a relation:

```
  reltype = SET OF elementtype ||
                WHERE rel IN reltype ==>
                  ALL r1,r2 IN rel
                    (r1.key=r2.key ==> r1=r2).
```

The key constraint is essential to relational data modelling since only unique keys can serve as element identifiers as required, for example, for the construction of higher relationships between elements. Relational languages, therefore, directly support the above class of annotated set-type definitions by a data structure **relation** that allows for type definitions equivalent to the previous one:

```
  reltype = RELATION key OF elementtype.
```

For each assignment of a relational expression, rex, to a variable, rel, of reltype, the relational type checker has to perform a test equivalent to

```
  IF ALL x1,x2 IN rex
        ( x1.key=x2.key ==> x1=x2 )
  THEN  rel:=rex
  ELSE  <exception> .
```

2.3. Predicative Support for Relations: Selectors and Constructors

The key constraint is, of course, not the only condition one would like to have maintained automatically on a database. Take, for example, a relation NorthSouth_0 containing city pairs such that the first element is located immediately to the north of the second one, i.e. with no city in between:

```
TYPE citytype = RECORD
                    cityid: cityidtype;
                    size: integer;
                    ...
                END;

     cityrel = RELATION cityid OF citytype;
     northsouthrel =
           RELATION north,south OF
              RECORD
                  north,south: cityidtype
              END;
VAR Cities: cityrel;
    NorthSouth_0 : northsouthrel .
```

For example, NorthSouth_0 may contain the following tuples:

```
             north        south
             --------------------
             Oslo         Flensburg
             Flensburg    Munich
             Stockholm    Gdansk
             Gdansk       Vienna
             Helsinki     Wilna
             Munich       Rome
```

Since the attributes, north and south, of the NorthSouth_0 relation are supposed to relate cities, they have to refer to elements in the relation Cities. The corresponding referential integrity constraint can be expressed by annotating the type of the NorthSouth_0 relation:

```
VAR NorthSouth_0: northsouthrel ||
      WHERE r IN NorthSouth_0 ==>
         SOME r1,r2 IN Cities
            (r.north=r1.cityid) AND
               (r.south=r2.cityid).
```

In a relational language such a constraint can be enforced by a conditional which controls assignment of relational expression, rex, to the NorthSouth_0 relation:

```
IF ALL x IN rex
     (SOME r1,r2 IN Cities
          (x.north=r1.cityid AND
           x.south=r2.cityid) )
THEN NorthSouth_0:=rex
ELSE <exception>  .
```

In expecting frequent use of relations in such "conditional patterns", the database programming language DBPL [ScMa 83], [MaReSc 84] provides an abstraction mechanism for such patterns through the notion of a **selector**. Referential integrity on relations of type northsouthrel, for example, can be maintained by

```
SELECTOR refint FOR Rel: northsouthrel();
BEGIN  EACH r IN Rel:
          SOME r1,r2 IN Cities
             (r.north=r1.cityid AND
              r.south=r2.cityid)
END refint .
```

An assignment to a selected relation variable, for example,

```
NorthSouth_0 [refint] := rex,
```

is defined to be equivalent to the above conditional assignment to the full relation variable NorthSouth_0.

In summary, selectors "factor out" conditions on relations, represent them uniformally, and make them available to all database system components that have to reason about programs and data (such as query optimizer, concurrency manager, and integrity subsystem). The selector concept is illustrated in Fig. 1.

```
Fact Relation: +-------------+
          Rel  |             |
               |      |      |
               |      |      |
               |      v      |
              +|-------------|+ Rel[s]:
              ||             ||   Selected
              ||             ||   Relation
              +|-------------|+
               |      ^      |
               |      |      |
               |      |      |
               |             |
               +-------------+
```

Figure 1: Selectors and Relations

While selectors provide support when data elements are to be **excluded** from a relation there is also a need for supporting the contrary - when additional derived data objects are to be **included** into a relation.

For example, a relation, NorthSouth_1, can be defined that relates - based on the data in relation NorthSouth_0 - two cities if and only if there is at most one city in between.

An annotated definition of relation NorthSouth_1 would read as follows:

```
VAR NorthSouth_1: northsouthrel ||
   WHERE (r IN NorthSouth_0 ==>
               r IN NorthSouth_1)
     AND (r1,r2 IN NorthSouth_0 ==>
               (r1.south=r2.north ==>
                  <r1.north,r2.south>
                     IN NorthSouth_1)) .
```

In a relational language the value of such a relation, NorthSouth_1, can be denoted by a query expression in terms of predicates over the NorthSouth_0 relation:

```
northsouthrel
   { EACH r IN NorthSouth_0: TRUE,
     <r1.north,r2.south> OF
         EACH r1,r2 IN NorthSouth_0:
                r1.south=r2.north  }.
```

In anticipating the frequent use of relations in such "expressional patterns" this paper proposes an abstraction mechanism for such patterns based on the notion of a **constructor.**

As an example, the northsouth_1-relationship based on relations of type northsouthrel can be constructed by

```
CONSTRUCTOR northsouth_1
   FOR Rel:northsouthrel (): northsouthrel;
BEGIN   EACH r IN Rel: TRUE,
        <r1.north,r2.south> OF
            EACH r1,r2 IN Rel:
               r1.south=r2.north
END northsouth_1.
```

The value of a constructed variable, for example,

```
NorthSouth_0 {northsouth_1}
```

is defined to be equal to the value of the above relational expression of type northsouthrel.

For the above example value of NorthSouth_0, this constructor application constructs the pairs:

north	south
Oslo	Flensburg
Flensburg	Munich
Stockholm	Gdansk
Gdansk	Vienna
Helsinki	Wilna
Munich	Rome
Oslo	Munich
Flensburg	Rome
Stockholm	Vienna

In the same sense that selectors isolate the constraints imposed on selected relations, constructors factor out the rules that define the elements in constructed relations. The idea is illustrated in Fig. 2.

```
Constructed     +-------------+
Relation Rel{c} |             |
                |      ^      |
                |      |      |
                |      |      |
               +|-------------|+ Fact
               ||             ||  Relation:
               ||             ||   Rel
               +|-------------|+
                |      |      |
                |      |      |
                |      v      |
                |             |
                +-------------+
```

Figure 2: Constructor and Relations

In the subsequent section the basic issues of constructor semantics are discussed with emphasis on recursive constructor definition and constructor convergence.

3. Relation Constructors

In this section we discuss the notion of a constructor in more detail. We first provide some examples based on the relations introduced in section 2, and then define the semantics of recursive constructors formally. Constructors are then compared with other approaches to rule and fact management.

3.1. Recursive Constructors

The above simple constructor, northsouth_1, representing all city pairs with at most one city in between on the way south, can be generalized to a sequence of constructors, northsouth_n, representing all pairs of cities with at most n cities in between on the way south.

```
CONSTRUCTOR northsouth_n
   FOR Rel:northsouthrel(): northsouthrel;
BEGIN EACH r IN Rel: TRUE,
      <r1.north,r2.south> OF
         EACH r1 IN Rel,
           EACH r2 IN Rel (northsouth_n-1):
                  (r1.south=r2.north)
END northsouth_n .
```

For the definition of a constructor, northsouth, representing all city pairs separated by an arbitrary number of cities on the way south, we utilize **simple recursion**:

```
CONSTRUCTOR northsouth
   FOR Rel:northsouthrel(): northsouthrel;
BEGIN EACH r IN Rel: TRUE,
      <r1.north,r2.south> OF
         EACH r1 IN Rel,
           EACH r2 IN Rel(northsouth):
                  (r1.south=r2.north)
END northsouth.
```

Intuitively, the value of a constructed relation

```
NorthSouth_0 (northsouth)
```

can be seen as the limit of the sequence of constructor applications

```
NorthSouth_0 (northsouth_n)
```

The details of constructor semantics are given in section 3.2.

Since the sequence in our example is monotonic, the limit exists and can be implemented by a finite loop using a relation variable, NorthSouth:

```
NorthSouth := {};
REPEAT
   OldNorthSouth := NorthSouth;
   NorthSouth    :=
     {EACH r IN NorthSouth_0:TRUE,
      <r1.north,r2.south> OF
         EACH r1 IN NorthSouth_0,
           EACH r2 IN OldNorthSouth:
                      r1.south=r2.north   }
UNTIL OldNorthSouth = NorthSouth .
```

Assuming that the variable NorthSouth_0 contains the same pairs as given in section 2.3, the constructor application

```
NorthSouth_0(northsouth)
```

constructs the following pairs:

```
north      south
-------------------
Oslo       Flensburg
Flensburg  Munich
Stockholm  Gdansk
Gdansk     Vienna
Helsinki   Wilna
Munich     Rome

Oslo       Munich
Flensburg  Rome
Stockholm  Vienna

Oslo       Rome
```

The relational expression

```
{<r.south> OF
 EACH r IN NorthSouth_0(northsouth):
                    r.north="Stockholm"}
```

computes the set

```
{Gdansk, Vienna}
```

To give an example of **mutual recursion**, we introduce a second relation between cities containing all city pairs such that the first element is located immediately to the west of the second one:

```
TYPE westeastrel  =
       RELATION west,east OF
            RECORD
                 west,east: cityidtype
            END;
VAR WestEast_0: westeastrel .
```

We are interested in the cities which can be reached by first going south and by first going east (for the sake of simplicity, we ignore the directions north and west). We define two mutually recursive constructors for this task, namely southfirst and eastfirst:

```
CONSTRUCTOR southfirst
   FOR Rel: northsouthrel
     (Param:westeastrel):northsouthrel;
BEGIN
  EACH r IN Rel: TRUE,
  <r1.north,r2.south> OF
     EACH r1 IN Rel,
       EACH r2 IN Rel (southfirst(Param)):
                 r1.south=r2.north,
  <r1.north,r2.east> OF
     EACH r1 IN Rel,
       EACH r2 IN Param (eastfirst(Rel)):
                 r1.south=r2.west
 END southfirst;
```

```
CONSTRUCTOR eastfirst
   FOR Rel: westeastrel
      (Param:northsouthrel):westeastrel;
BEGIN
  EACH r IN Rel: TRUE,
  <r1.west,r2.east> OF
     EACH r1 IN Rel,
       EACH r2 IN Rel {eastfirst(Param)}:
                 r1.east=r2.west,
  <r1.west,r2.south> OF
     EACH r1 IN Rel,
       EACH r2 IN Param {southfirst(Rel)}:
                 r1.east=r2.north
END eastfirst .
```

We can apply both constructors to the relations NorthSouth_O and WestEast_O as follows:

```
NorthSouth_O {southfirst(WestEast_O)}
    and
WestEast_O {eastfirst(NorthSouth_O)} .
```

The values of these mutually recursive constructed relations are defined by the limits of mutually defined sequences; again, the details are given in section 3.2.

Since the sequences are monotonic, the limits exist and can be implemented by the following loop using the auxiliary variables, Southfirst and Eastfirst, for the values of the constructed relations:

```
Eastfirst:={}; Southfirst:={};
REPEAT
   Oldeast:=Eastfirst;
   Oldsouth:=Southfirst;
   Southfirst:=
     southfirst_fct(Oldsouth,Oldeast);
   Eastfirst :=
     eastfirst_fct( Oldsouth,Oldeast)
UNTIL Oldeast=Eastfirst AND
        Oldsouth=Southfirst .
```

southfirst_fct and eastfirst_fct are relation-valued functions based on the definition of the constructors, southfirst and eastfirst.

For example, the variable WestEast_O is defined as

east	west
Oslo	Stockholm
Stockholm	Helsinki
Helsinki	Leningrad
Flensburg	Gdansk
Gdansk	Wilna
Munich	Vienna

.

NorthSouth_O is defined as in section 2.3 .

The application of the constructors gives the following relations:

southfirst:

north	south
Oslo	Flensburg
Flensburg	Munich
Stockholm	Gdansk
Gdansk	Vienna
Helsinki	Wilna
Munich	Rome
Oslo	Munich
Flensburg	Rome
Stockholm	Vienna
Oslo	Gdansk
Flensburg	Vienna
Stockholm	Wilna
Oslo	Rome
Oslo	Vienna
Oslo	Wilna

eastfirst:

east	west
Oslo	Stockholm
Stockholm	Helsinki
Helsinki	Leningrad
Flensburg	Gdansk
Gdansk	Wilna
Munich	Vienna
Oslo	Helsinki
Stockholm	Leningrad
Flensburg	Wilna
Oslo	Gdansk
Stockholm	Wilna
Flensburg	Vienna
Oslo	Leningrad
Oslo	Wilna
Oslo	Vienna

If we want to start in a particular city, we can write the corresponding selectors as follows:

```
SELECTOR southfrom
  FOR Rel:northsouthrel (city:cityidtype);
BEGIN
    EACH r IN Rel: r.north = city
END southfrom;

SELECTOR eastfrom
  FOR Rel:westeastrel (city:cityidtype);
BEGIN
    EACH r IN Rel: r.west = city
END eastfrom.
```

We can apply these selectors by

```
NorthSouth_O {southfirst(WestEast_O)}
    [southfrom("Oslo")]
```

giving the pairs

north	south
Oslo	Flensburg
Oslo	Munich
Oslo	Gdansk
Oslo	Rome
Oslo	Vienna
Oslo	Wilna

and

```
WestEast_O {eastfirst(NorthSouth_O)}
    [eastfrom("Flensburg")]
```

giving the pairs

west	east
Flensburg	Gdansk
Flensburg	Wilna
Flensburg	Vienna

.

In most applications it is obvious to which relation a constructor is to be applied (for example, NorthSouth_O) and which relations are to serve as arguments (for example, WestEast_0). In a few cases, however, this choice may be difficult and the programmer may prefer to start with an empty relation (for example, if the constructor is based on a join of several base relations rather than growing out of a single one).

3.2. Formal Constructor Semantics

In general, a database program may contain a large number, m, of mutually dependent constructors:

```
CONSTRUCTOR $c_1$
      FOR $Rel_1$: $reltype_1$ (...): $resulttype_1$;
BEGIN
   $f_1$ (..., $applyc_{1,1}$, ..., $applyc_{1,n1}$)
END $c_1$;
...
CONSTRUCTOR $c_m$
      FOR $Rel_m$: $reltype_m$ (...): $resulttype_m$;
BEGIN
   $f_m$ (..., $applyc_{m,1}$, ..., $applyc_{m,nm}$)
END $c_m$ .
```

where each $applyc_{i,j}$ is a (possibly recursive) constructor application of the form Rel {c(...)}. Rel is a relation name known in the context of f_i, and c is one of our c_i. $\{f_i\}$ is a relational calculus expression. To simplify indexing, we rename our constructor applications $applyc_{i,j}$ to $apply_1, ..., apply_l$, $l = n_1 + ... + n_m$.

We impose the following restriction on the recursive constructor applications in the constructor definitions: The parameters $p_1, ..., p_n$ in the constructor application Rel{c(p_1, ...,p_n)} are not allowed to depend on the recursion. This means that the constructor applications remain the same throughout the recursion.

The semantics of a constructor application

$apply_0$ = Actrel { c_i (...)},

on an actual relation Actrel, is defined as follows:

We construct $l+1$ functions

g_0($apply_0$, $apply_1$, ..., $apply_l$)
...
g_l($apply_0$, $apply_1$, ..., $apply_l$);

function g_j is constructed by taking the function f_p which corresponds to the constructor in the application $apply_j$, and replacing all formal parameters by their actual values.

We define

$apply_{i,0} = \{\}$ $\quad$ (i=0,1,...,l)
$apply_{i,k+1} = g_i(apply_{0,k}, ..., apply_{l,k})$

and compute the limits:

$apply_i = \lim_{k \to \infty} apply_{i,k}$.

The value of constructor application

Actrel { c_i (...) }

is given by $apply_0$.

Of course this definition makes sense only if the limit of the above sequences exists. If the functions f_i are monotonic, we have $apply_{i,0} \subseteq apply_{i,1}$, and therefore, by induction, $apply_{i,k} \subseteq apply_{i,k+1}$. Because all relations are based on finite domains, there must be a step j such that $apply_{i,j} = apply_{i,j+1}$. If, therefore, the f_i are monotonic, the limits exist and are reached after a finite number of steps. It can be shown [ChHa 82] that the functions f_i are monotonic if their predicates are free of negation and universal quantifiers.

Note that, according to [AhUl 79] and [Tars 55], we compute the least fixed point of the system of equations

$apply_0 = g_0 (apply_0, ..., apply_l)$
...
$apply_l = g_l (apply_0, ..., apply_l)$.

A program for computing the limits can be written in the same way as for our examples in 3.1.

3.3. Negation and Universal Quantification

Database languages such as DBPL and Pascal/R [Schm 77] allow universal quantification of element variables as well as negation of relational predicates. However, constructors containing negation and universal quantification may be meaningless because the limit of the fixed point computation may not exist, as, for example, in

```
CONSTRUCTOR nonsense
   FOR Rel: anytype (): anytype;
BEGIN  EACH r IN Rel:
            NOT (r IN Rel{nonsense})
END nonsense.
```

The iteration yields

{}
Rel
{}

and has obviously no limit.

There are, however, meaningful constructor definitions with negation and universal quantification, and the DBPL compiler will recognize a subclass thereof, defined by the so-called positivity constraint. Let us start with auxiliary definitions:

Definition: Names appearing under NOT and ALL

Let f be a DBPL expression.

A name n is said to **appear under ALL** if f is of the form
 f = ... ALL r IN exp (p(r,...)) ...
and n appears in exp.

A name n is said to **appear under NOT** if f is of the form
 f = ... NOT fact ...
and n appears in the subexpression fact.

Note that these definitions may be nested, i.e., a name may appear under several ALLs and NOTs. In
ALL r IN exp (p(r,...))
a name n appearing in p(r,...) but not in exp is not considered to appear under this ALL.

Definition: positivity of a DBPL expression

Let f(Rel_1, ..., REL_n) be a DBPL expression.
f is said to satisfy the **positivity constraint** if each occurrence of Rel_i appears under an **even** total number of negations and universal quantifiers.

The idea of positive expressions is similar to 'safe' expressions in [Ullm 82] by which the definition of infinite relations in relational calculus expressions is avoided.

Lemma:

Each DBPL expression f(Rel_1, ..., Rel_n) that satisfies the positivity constraint is monotonic in all its arguments.

Proof Sketch:

Change f as follows: Replace range-coupled quantifiers by their one-sorted version [JaKo 83]:
ALL r IN Rel (pred(r,...)) =
ALL r (NOT(r IN Rel) OR pred(r,...))
SOME r IN Rel (pred(r,...)) =
SOME r (r IN Rel AND pred(r,...))
The result is that we have replaced each occurrence of Rel_i under a universal quantifier by an occurrence under NOT. Thus, if the number of ALLs plus the number of NOTs over each occurrence of Rel give an even total, we now have an even number of NOTs over each occurence Rel of a Rel_i. However, if this is the case, we can remove the negations, using generalized deMorgan and distribution laws to move all NOTs as far into the expression (i.e. to the right) as possible and applying the double negation law NOT(NOT(pred)=pred. The resulting expression will be monotonic in all its arguments.

A similar lemma is given in [ChHa 82]. For simplicity, the DBPL compiler accepts only constructors satisfying the positivity constraint. It should be noted, however, that there are non-monotonic constructors for which the limit of the fixed point computation exists. The following example is derived from [Hehn 84]:

```
TYPE cardrel = RELATION ... OF
               RECORD number: CARDINAL END;
CONSTRUCTOR strange
  FOR Baserel: cardrel (): cardrel;
BEGIN EACH r IN Baserel:
        NOT SOME s IN Baserel {strange}
             (r.number=s.number+1)
END strange .
```

Let Rel = {0, 1, 2, 3, 4, 5, 6}. The computation of Rel {strange} through the iteration

{}
{0,1,2,3,4,5,6}
{0}
{0,2,3,4,5,6}
{0,2}
{0,2,4,5,6}
{0,2,4}
{0,2,4,6}
{0,2,4,6}
etc.

has the limit {0,2,4,6}.

Examples like this one, however, look artificial and are much more difficult for the programmer and compiler to understand than the simple positivity constraint; they are, therefore, not allowed in DBPL.

3.4. Options for Fixpoint Enhancements in Database Programming

In this subsection we summarize the options for expressing the Least Fixpoint Operator semantics in a database programming language like DBPL. For database programming languages we distinguish six possibilities to include fixpoint operations. Our constructor approach can be seen as the seventh alternative.

- Programm iteration;
- Recursive boolean functions and procedures;
- Specialized LFP operators;
- Equational relation variable declarations;
- Views as relation-valued functions;
- Logic Programming.

The first two options have long been available in early languages such as Pascal/R [Schm 77] although they have not received much attention there. The programs for computing the limits in section 3.1 may serve as examples of this approach. Similar effects can also be achieved using recursive functions (to generate recursive relations or to test membership recursively). Both methods share the problem of too much generality since the programmer can write anything into the loop or the function body; this severely limits query optimization. Moreover, the end of the loop or the recursion has to be programmed manually which gives us the problem of infinite loops and infinite recursion.

Query-by-example [Zloo 77] was one of the first systems to contain a **specialized operator** for transitive closure. More recently, the query language QUEL has been augmented with an operator * which can extend any QUEL command with the semantics "to repeat the command forever" [Kung 84], [IoShWo 84]. [EmEmDo 84] combine a similar approach with view-oriented concepts as described below. While some algebraic optimization of such language extensions is possible [Kung 84], the approach is essentially procedural and does not seem to fit well into a calculus-oriented language.

Equational relation definition bears a close resemblance to relation definition by constructors. However, instead of constructing relations explicitly from conventionally typed variables, the type concept itself can be extended to allow implicit relation definition by using a set of constraining conditions:

```
VAR NorthSouth_0: northsouthrel;
    NorthSouth:
      northsouthrel
        { EACH r IN NorthSouth_0: TRUE,
          <r1.north,r2.south> OF
            EACH r1 IN NorthSouth_0,
            EACH r2 IN NorthSouth:
                   r1.south=r2.north } .
```

The work on equational constraint expressions [Morg 84] follows a similar approach.

A number of researchers have proposed parameterized view definitions for query language extensions (e.g., [MaReSc 84], [EmEmDo 84]). From a programming language standpoint, views can be interpreted in two different ways. If relations are considered as generalized tables or arrays, these structures seem to be adequately handled by selectors and constructors. If relations are considered as sets, views can be considered as **relation-valued functions**. Since recursive functions are available in modern programming languages, the extension to relation-valued functions would be small, for example:

```
FUNCTION northsouth
    (Current:northsouthrel): northsouthrel;
VAR New: northsouthrel;
BEGIN
   New := { EACH r IN Current: TRUE,
            <c.north,d.south> OF
            EACH c,d IN Current:
                   c.south=d.north };
      IF New = Current
      THEN RETURN Current
      ELSE RETURN northsouth(New)
END northsouth;
...
NorthSouth := northsouth(NorthSouth_0).
```

However, as previously discussed, functions are too general to be optimized efficiently. Of course, if used in a pure query language environment such as SQL, relation-valued functions can be restricted to only define parameterized views and thus may not raise the problems present in tightly integrated database programming languages.

One of the most important areas closely related to our work is that on **logic programming** as exemplified by PROLOG (e.g. [ClMe 81]). Based on Horn clauses, the programming language PROLOG (without cut, fail and negation) can be shown to be equivalent to a data base query language with the least fixed point operator [ChHa 82]. As far as the language extensions proposed in this paper are concerned, we have the following lemma:

Lemma:

The constructor mechanism is as powerful as function-free PROLOG without cut, fail, and negation.

Proof sketch: Horn clauses are precisely representable by applying a single fixed point operator to a positive existential query [ChHa 82]. Furthermore, mutual recursion can be replaced by a single fixed point operator by moving the mutual recursion into the arguments [AhUl 79]. Any query representable in function-free Horn clauses, therefore, is also representable by the constructor mechanism.

As far as negation is concerned, our approach assumes a closed world [Reit 78] and is guaranteed to terminate because of positivity. It is not, therefore, directly comparable with PROLOG's NOT. However, it seems to be more practical because the problem of infinite loops is eliminated.

4. Compilation and Optimization of Constructors

In this section we investigate the implementation of constructors and the optimization of queries in which constructed relations appear. Constructed relations are interpreted as a generalization of the range-nested expressions of [JaKo 83]. First, we study the compilation of queries over constructed relations into queries over base relations; certainly the most interesting part of this is the handling of recursion. Then we discuss the optimization of such queries. Rather than adding to the long list of specialized techniques for recursion optimization, we present a three-level framework tailored to the database programming environment in which such techniques can be integrated. For space reasons, details must be left to a forthcoming paper.

[JaKo 83] introduced a concept of range nesting for relational calculus expressions. Basically, it allows the substitution of relational expressions for range relations in queries using the following rules:

```
N1: {EACH r IN R: pred1 AND pred2}
          <==>
    {EACH r IN {EACH r' IN R: pred1}: pred2}
N2: SOME r IN R (pred1 AND pred2)
          <==>
    SOME r IN {EACH r' IN R: pred1} (pred2)
N3: ALL r IN R (NOT(pred1) OR pred2)
          <==>
    ALL r IN {EACH r' IN R: pred1} (pred2)
```

Selected and constructed relations can be interpreted as methods to **name** such extended range expressions. If we want to follow the <==

direction in order to understand and optimize a query in terms of base relations, the question becomes by which predicate pred1 to replace the constructed relation. Consider the expression

```
{EACH r IN Rel{constr}: pred(r)}
```

Clearly, the easiest solution is to compute Rel{constr} exhaustively by all least fixed points of related constructor definitions and then test pred(r). However, propagating the constraints given by pred(r) into the constructor definition may considerably reduce query evaluation costs. A case-by-case analysis of various constructor types will demonstrate how this can be done. Assume first that the definition of constr does **not** contain any constructed variable, i.e. constr works only on base relations.

Case 1 (Selector): The constructor definition contains a single relational expression (no union) with a single free variable. In this case the transformation rules N1 to N3 apply directly, possibly in conjunction with a projection on the target attributes.

Case 2 (Join): The constructor definition contains a single relational expression but possibly more than one variable. In this case substitute r.f in pred(r) by x.g if x.g appears in the position f of the constructor's target list (possibly with renaming).

Case 3 (Union): The constructor definition is a union of relational expressions. If pred(r) satisfies the positivity constraint, treat each of these relational expressions separately and let the result be the union of the expression values.

If the tuple variable whose range expression is constructed is existentially or universally quantified, the above rules apply in a similar fashion, corresponding to rules N2 and N3. The rules actually present just a minor generalization of [Ston 75].

Consider now the case that the constructor definition does contain constructed relations. The naive application of the above rules would give an infinite derivation sequence in the case of recursive constructors. Adapting a strategy described in [Naqv 84], [Venk 84], a finite representation of this infinite sequence can be devised from which appropriate least fixed point computations can be generated. Due to space limitations, we can only sketch the algorithm here, using the constructor northsouth as an example.

1. Augment each constructor definition by introducing a new tuple variable ranging over the result relation of the constructor:

```
EACH res IN Rel {northsouth}:
  SOME r IN Rel (r=res) OR
  SOME r1 IN Rel,SOME r2 IN Rel{northsouth}
      (r1.north=res.north) AND
      (r2.south=res.south) AND
      (r1.south=r2.north)
```

2. Construct a quant graph for each thus augmented expression. A quant graph represents a relational calculus query [JaKo 83]; it has a node for each tuple variable with its range definition and a directed arc in quantifier direction (outside in) for each join term and each enforced quantifier sequence.
3. The reader may have noticed that, as stated, the above expression and quant graph are not yet equivalent to the previously defined constructor semantics since they ignore the distinction between the two different occurrences of Rel{northsouth}; i.e., the range relation of the variable res is one recursion step further than the range relation of r2. (Indeed, [AhUl 79] shows that an equivalent pure relational calculus expression cannot exist). To express this relationship, we construct directed arcs from each quantified node with a constructed range relation (in the example: r2) to the corresponding constructor definition (i.e. res in the example). We have now constructed the equivalent of a clause interconnectivity graph [Sick 76]. The extended quant graph for the above example is given in Fig. 3.
4. Evaluate each component as follows. For acyclic subgraphs, replace the constructor definitions by subqueries on base relations and optimize as described, e.g., in [Jark 84]. Most cyclic subgraphs correspond to recursion (for exceptions such as tautologies see [Sick 76]). We can now apply any standard algorithm, i.e., LFP computation of the related constructor definitions, recursive calls of iterative procedures [HeNa 84], or tuple-at-a-time cycling [McSh 81]; or we can attempt to employ capture rules [Ullm 84] to detect special cases such as those described in [Schn 78], [MiNi 83], [Fron 84].

Applying this method at query evaluation time may be quite expensive if many constructed relations are defined. Our optimization strategy tries to move many of these tasks into the compilation phase; this is even more important in a database programming language than in an interactive query language because compilation is usually decoupled from execution.

```
           +-------------------------------+
           | EACH res IN Rel {northsouth}  |<-----------+
           +-------------------------------+            |
             |                |_____________|           |
       r     |     res.north  |     AND     | res.south |
       =     |         =      |             |     =     |
      res    |     r1.north   |             | r1.south  |
             V                V             V           |
      +---------+      +----------+          +-----------------+
      | SOME r  |      | SOME r1  | r1.south=| EACH r2 IN      |
      | IN Rel  |      | IN Rel   |--------->| Rel(northsouth) |
      +---------+      +----------+ r2.north +-----------------+
```

Figure 3: Extended quant graph

On the other hand, database programming languages are frequently used to implement higher-level interfaces and, therefore, contain only incompletely specified query forms rather than full queries. These observations lead to a three-level strategy in the optimization of the system that makes full use of the degrees of information available to different phases of the DBPL compiler and to the run time support system.

On the **type checking** level the compiler performs an analysis of the individual constructor definitions and their relationships. For example, this phase contains the positivity test within the constructor definition. It also constructs a rough version of the extended quant graphs described above. In terms of optimization, one major purpose of this is to partition the set of constructor definitions in disconnected subgraphs that can always be processed separately.

This partitioning can be done by stepwise refinement. A first version of the graph would mention relation and constructor names alone. If some of the remaining partitions are still very large, they could then be refined to an intermediate level that, e.g., distinguishes between free and bound variables [Ullm 84].

On the **query compilation** level the compiler looks at the query forms appearing in the database program. These query forms may use range relations that apply constructors to base relations, selected relations, or constructed relations. The compiler can now instantiate the appropriate constructor definition graphs and complete the construction of full extended quant graphs for each query. If such a graph contains a recursive cycle, the compiler can generate an appropriate version of the fixed point algorithm [HeNa 84], [Ullm 84]. For nonrecursive queries, full compilation and optimization are performed.

Recursion optimization can be based on the algorithms given in section 3.1. using Bayer's Delta-Transformation concept [Baye 85]. The main idea is to use loops for implementing recursion which work only on the increments of the recursively defined relations in each iteration. Of course, this is not always possible, e.g., in case the join of two recursively defined relations is to be performed in another recursively defined relation. [Baye 85] shows, however, that even then it is feasible to extract precisely those subexpressions that may produce new values for the recursively defined relations.

In the southfirst and eastfirst constructors of section 3.1, only the increments are needed to compute the least fixed points, stored in variables Deltasouth and Deltaeast:

```
Southfirst := NorthSouth_0;
Eastfirst  := WestEast_0;
Deltasouth := Southfirst;
Deltaeast  := Eastfirst;
REPEAT
   Hsouth := {<r1.north,r2.south> OF
                   EACH r1 IN NorthSouth_0,
                      EACH r2 IN Deltasouth:
                           r1.south=r2.north,
                 <r1.north,r2.east> OF
                   EACH r1 IN NorthSouth_0,
                      EACH r2 IN Deltaeast:
                           r1.south=r2.west};
   Heast  := {<r1.west,r2.east> OF
                   EACH r1 IN WestEast_0,
                      EACH r2 IN Deltaeast:
                           r1.east=r2.west,
                 <r1.west,r2.south> OF
                   EACH r1 IN WestEast_0,
                      EACH r2 IN Deltasouth:
                           r1.east=r2.north};
   Deltasouth := Hsouth;
   Deltaeast  := Heast;

   union(Southfirst,Deltasouth,bool1);
   union(Eastfirst, Deltaeast, bool2)

UNTIL bool1 AND bool2,
```

where the procedure

```
union(rel,deltarel,bool)
```

is defined as:

```
bool:= (deltarel ⊆ rel);
rel := (rel U deltarel).
```

Note that in this implementation we cannot use the condition

```
(Deltasouth={}) AND (Deltaeast={})
```

as the terminating condition because there may be tuples that are generated over and over again, for example, if, say, one of the base relations contains a cycle. The loop, therefore, terminates if both Delta-relations do not contain new tuples.

Thus far, we ignore that constructor and selector definitions may contain parameters. In the case where these are constant values in restrictive terms of the constructor definition or associated query, we can represent this situation by defining an appropriate selector. This selector will provide a logical or even physical access path for instantiations of the parameters. A logical access path is a compiled procedure with dummy constants [HeNa 84]. A physical access path actually materializes a relation corresponding to the query with the constants used as variables and partitions it according to the different constant values. Clearly a physical access path would be generated only in case of heavy query usage since unrestricted constructed relations may be very large. Maintenance of such access paths also becomes very expensive [ShTZ 84].

If the parameters are of type relation, they may be instantiated at run time with constructed relations, possibly leading to a connection among previously independent subgraphs. At compilation time, this case will only permit partial logical access paths to be generated.

Finally, the **run time support** subsystem of query processing must help in the evaluation of fully instantiated queries. In some cases this will mean simply the execution of the compiled database programs. In the case of selectors generated at compile time, physical access paths may be generalized and utilized. In the case of relation parameters, it may mean the integration of pieces of precompiled definitions into meaningful database programs. A major advantage of the DBPL environment over, say, a PROLOG environment is that all of these tasks can be formulated elegantly with the existing language tools and are executed in a set-oriented constructive fashion rather than by tuple-oriented theorem proving.

5. Conclusion

Relational database systems are based on first-order logic and provide, within that framework, solutions for many technical problems with data-intensive applications, such as query optimization, concurrency management, and data distribution. While AI-oriented systems have traditionally emphasized issues of knowledge representation and reasoning, their future applications will require database support for problems originating from large-scale fact and rule management.

We argue that the DBMS should remain responsible for as much efficient mass-processing of data as possible, whereas the AI system should retain the responsibility for the more subtle tasks, such as handling open worlds (i.e., incomplete knowledge and non-monotonic reasoning) for which intelligent and frequently problem-specific heuristics are needed since the problem in general is computationally intractable or even undecidable [BrLe 84]. The proposed extension of the relational approach handles nested and recursive rule definition and evaluation adequately and efficiently. In an orthogonal approach to data model extension we investigate object structures that allow nested and recursive structure definition and component selection [Lame 84], [LaMuSc 84], [ScLi 85].

In [LiScJa 85] we demonstrate how recursive data structures can be used for constructor representation, thus allowing for the definition, update, and querying of large constructor bases. These approaches are considered as first steps towards integrated fact and rule base management utilizing advanced relational database technology.

References

[AhUl 79]
Aho,A.V.; Ullman,J.D.: Universality of Data Retrieval Languages, 6th ACM Symp. on Principles of Programming Languages, San Antonio, Texas, January 1979

[Baye 85]
Bayer,R.: Query Evaluation and Recursion in Deductive Database Systems, Institut für Informatik Technische Universität München January 1985

[BrLe 84]
Brachman,R.; Levesque,H.J.: What Makes a Knowledge Base Knowledgable - A View of Databases from the Knowledge Level, in [Kers 84], 30-39

[BrMySc 84]
Brodie,M.L.; Mylopoulos,J.; Schmidt,J.W. (eds.): On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer Verlag, 1984

[ChHa 82]
Chandra,A.K.; Harel,D.: Horn Clauses and the Fixpoint Query Hierarchy, ACM Symposium on Principles of Database Systems, Los Angeles, 1982, 158-163

[ClMe 81]
Clocksin,W.F.; Mellish,C.S.: Programming in

PROLOG, Springer Verlag 1981

[Codd 72]
Codd,E.F.: Relational Completeness of Data Base Sublanguages, in R. Rustin (ed.): Data Base Systems, Prentice Hall, Englewood Cliffs, NJ, 1972, 65-98

[Covi 85]
Covington,M.A.: Eliminating Unwanted Loops in PROLOG, ACM SIGPLAN Notices 20,1 (Jan. 1985), 20-26

[Deut 81]
Deutsch,L.P.: Summary of Workshop Session on Types, in Brodie,M.L.; Zilles,S. (eds.): Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, SIGPLAN Notices, Vol. 16, No. 1, January 1981, p. 49

[EmEmDo 84]
van Emde Boas-Lubsen,H.; van Emde Boas,P.; Doedens,C.F.J.: Extending a Relational Database With Logic Programming Facilities, IBM INS-Development Center, TR 13.195, Uithorn, The Netherlands, 1984

[Fron 84]
Fronhoefer,B.: Heuristics For Recursion Improvement, Proc. 6th ECAI, Pisa, 1984, 577-580

[GaMN 84]
Gallaire,H.; Minker,J.; Nicolas,J.M.: Logic and Databases: A Deductive Approach, Comp. Surveys, Vol.16, No.2, June 1984, 153-185

[Gries 81]
Gries,D.: The Science of Programming, Springer Verlag, 1981

[Hehn 84]
Hehner,E.C.R.: The Logic of Programming, Prentice-Hall International, 1984

[HeNa 84]
Henschen,L.J.; Naqui,S.A.: On Compiling Queries in Recursive First-Order Databases, JACM Vol.31, No.1, January 1984, 47-85

[IoShWo 84]
Ioannidis,Y.; Shinkle,L.L.; Wong,E.: Enhancing INGRES with Deductive Power, in [Kers 84], 847-850

[Jark 84]
Jarke,M.: External Semantic Query Simplification: A Graph-theoretic Approach and Its Implementation in PROLOG, in [Kers 84]

[JaKo 83]
Jarke,M.; Koch,J.: Range Nesting: A Fast Method to Evaluate Quantified Queries, Proc. ACM SIGMOD Conf., San Jose, Ca, 196-206

[JaVa 84]
Jarke,M.; Vassiliou,Y.: Coupling Expert Systems and Database Management Systems, in Reitman,W.R. (ed.): Artificial Intelligence Applications for Business, Ablex,Norwood,NJ, 65-85

[Kers 84]
Kerschberg,L. (ed.): Proceedings of the First International Workshop on Expert Database Systems, Kiawah Island, South Carolina, October 1984

[Krie 84]
Krieg-Brueckner,B.: Types in the Programming Language Ada, in [BrMySc 84], 385-410

[Kung 84]
Kung,R.M.; Hanson,E.; Ioannadis,Y.; Sellis,T.; Shapiro,L.; Stonebraker,M.: Heuristic Search in Data Base Systems, in [Kers 84], 96-107

[Lame 84]
Lamersdorf,W.: Recursive Data Models for Non-Conventional Database Applications, Proc. Intern. IEEE Conference on Data Engineering, Los Angeles, April 1984

[LaMuSc 84]
Lamersdorf,W.; Müller,G.; Schmidt,J.W.: Language Support for Office Modelling, Proc. 10th VLDB Conf., Singapore, August 1984, 280-290

[LiScJa 85]
Linnemann,V.; Schmidt,J.W.; Jarke,M.: Integrated Fact and Rule Management Based on Relational Technology, Workshop on Knowledge Base Management Systems, Crete, Greece, June 24-26,1985

[MaReSc 84]
Mall,M.; Schmidt,J.W.; Reimer,M.: Data Selection, Sharing, and Access Control in a Relational Scenario, in [BrMySc 84], 411-436

[MaMaJo 84]
Marque-Pucheu,G.; Martin-Gallausiaux,J.; Jomier,G.: Interfacing PROLOG and Relational Data Base Management Systems, in G.Gardarin; E.Gelenbe (eds.): New Directions for Databases, Academic Press, 1984

[McSh 81]
McKay,D.P.; Shapiro,S.C.: Using Active Interconnectivity Graphs for Reasoning with Recursive Rules, Proc. 7th IJCAI, Vancouver,BC, 368-374

[MiNi 83]
Minker,J.; Nicolas,J.-M.: On Recursive Axioms in Deductive Databases, Inform. Systems Vol. 8, No.1, 1983, 1-13

[Morg 84]
Morgenstern,M.: Constraint Equations: Declarative Expression of Constraints With Automatic Enforcement, Proc. 10th VLDB Conf., Singapore, August 1984, 291-300

[MyBr 85]
Mylopoulos,J.; Brodie,M.L.: AI and Databases: Semantic Versus Computational Theories of Information Systems, in Ariav,G.; Clifford,J. (eds.): New Directions

for Database Systems, Ablex, Norwood, N.J. 1985

[Naqv 84]
Naqvi,S.A.: PROLOG and Relational Databases. A Road to Data-intensive Expert Systems, in [Kers 84]

[Reit 78]
Reiter,R.: On Closed World Data Bases, in: Gallaire,H.; Minker,J.: Logic and Data Bases, Plenum Press 1978, 55-76

[Schm 77]
Schmidt,J.W.: Some High-level Language Constructs for Data of Type Relation, ACM TODS 2,3 (1977), 247-261

[Schn 78]
Schnorr,C.P.: An Algorithm for Transitive Closure With Linear Expected Time, SIAM Journal of Computing 7:2, 127-133

[ScLi 85]
Schmidt,J.W.; Linnemann,V.: Higher Level Relational Objects, British National Conference on Data Bases, Keele, Great Britain, July 1985

[ScMa 83]
Schmidt,J.W.; Mall,M.: Abstraction Mechanisms for Database Programming, Proc. SIGPLAN Symp. on Programming Language Issues in Software Systems, San Francisco, June 1983, 83-93

[ScWa 84]
Sciore,E.; Warren,D.S.: Towards an Integrated Database-PROLOG System, in [Kers 84], 801-814

[ShTZ 84]
Shmueli,O.; Tsur,S.; Zfirah,H.: Rule Support in PROLOG, in [Kers 84]

[Sick 76]
Sickel,S.: A Search Technique for Clause Interconnectivity Graphs, IEEE Transactions on Computers C25:8, 823-834

[Smith 81]
Smith,J.M.; Fox,S.; Landers,T.: Reference Manual for Adaplex, CCA, Cambridge, Mass., January 1981

[Smith 84]
Smith,J.M.: Expert Database Systems: A Perspective, in [Kers 84]

[Ston 75]
Stonebraker,M.: Implementation Of Integrity Constraints and Views By Query Modification, Proc. ACM SIGMOD Conf., San Jose, Ca, 1975, 65-78

[Tars 55]
Tarski,A.: A Lattice Theoretical Fixpoint Theorem and its Applications, Pacific J. Mathematics 5:2, June 1955, 285-309

[Ullm 82]
Ullman,J.D.: Principles of Database Systems, Computer Science Press, 2nd ed. 1982

[Ullm 84]
Ullman,J.D.: Implementation of Logical Query Languages for Databases, Report STAN-CS-84-1000, Stanford,Ca. 1984

[Venk 84]
Venken,R.: A PROLOG Meta-Interpreter for Partial Evaluation and Its Application To Source-To-Source Transformation and Query Optimization, Proc. 6th ECAI, Pisa, 1984, 91-100

[Wirth 83]
Wirth,N.: Programming in MODULA-2, Springer Verlag 1983

[Zani 84]
Zaniolo,C.: Prolog: A Database Query Language for All Seasons, in [Kers 84], 63-73

[Zloof 77]
Zloof,M.M.: Query-by-Example: a Database Language, IBM Syst. J. 16:4 (1977), 324-343