Integrated Fact and Rule Management Based on Database Technology

Joachim W. Schmidt, Lingyuan Ge [1]
Johann Wolfgang Goethe-Universität
Dantestrasse 9
D-6000 Frankfurt am Main 1
West Germany

Volker Linnemann
IBM Scientific Center Heidelberg
Tiergartenstraße 15
D-6900 Heidelberg
West Germany

Matthias Jarke
Universität Passau
Postfach 25 40
D-8390 Passau
West Germany

Abstract

Database programming languages integrate concepts of databases and programming languages to provide both, implementation tools for data-intensive applications and high-level user interfaces to databases. Frequently, database programs contain a large amount of application knowledge which is hidden in the procedural code and thus difficult to maintain with changing data and user views.
This chapter presents a first attempt to improve the situation by supporting the integrated definition and management of data and rules based on a set-oriented and predicative approach. For the definition of rules, we introduce and justify a new declarative language construct called constructor. Furthermore, we demonstrate how a Recursive Database Model can be used for constructor representation, thus allowing for the definition, update, and querying of large rule bases. The use of database technology for integrated fact and rule base management is shown to have some important advantages in terms of fact and rule integrity, question-answering, and explanation of results.

1 Introduction

Traditional database management systems (DBMSs) focus on the administration of large sets of formatted data at the 'symbol level' [BL 84]; they are rarely concerned with the interpretation of the stored data. To a certain degree the 'meaning' of data is defined by the data structures and integrity constraints of the database schema; however, much of the semantics is hidden in the application program code, or just in the users' heads. As a consequence, when compared to knowledge-based systems, DBMSs are weak in answering query and update requests intelligently or explaining unexpected or trivial results.

[1] Visiting scientist. Permanent address: Department of Electrical Engineering, Changsha Railway Institute, Changsha, Hunan, The People's Republic of China

A number of proposals have been made recently to endow DBMS with more deductive power. Some of these methods [KHI 84], [ISW 84] offer higher-level retrieval operators, thus extending the algebraic capabilities of query languages though not their linguistic flexibility (i.e., the capability of defining new terms). Another approach is based on extended view mechanisms; these can be added to the query language either by coupling them to a deductive programming language, such as Prolog [JV 84], or by integrating extended views into the language itself.
In [SL 85] and [JLS 85] we propose a declarative language construct called *constructor* that allows the definition of virtual, 'constructed' relations within the database programming language DBPL [SM 83], [MRS 84]. The constructor concept was shown to provide expressive power at the fixpoint level of the query language hierarchy given in [CH 82], equivalent to Prolog's [CM 81] function- and cut-free declarative semantics. However, as in Prolog itself, the rules thus defined are part of the program code and not easily changed or retrieved for rule management purposes. Our approach attacks this problem on two levels. At the conceptual level a type-oriented approach to rule definition is presented that provides a framework for rule bases with update and query capabilities (section 2). At the representational level the structures and operations for such a rule base are exemplified (section 3) by presenting a data-based implementation using a recursive database model [LAME 84], [LAME 85]. Some applications are outlined in section 4. The overall structure of this chapter is illustrated in Figure 1.

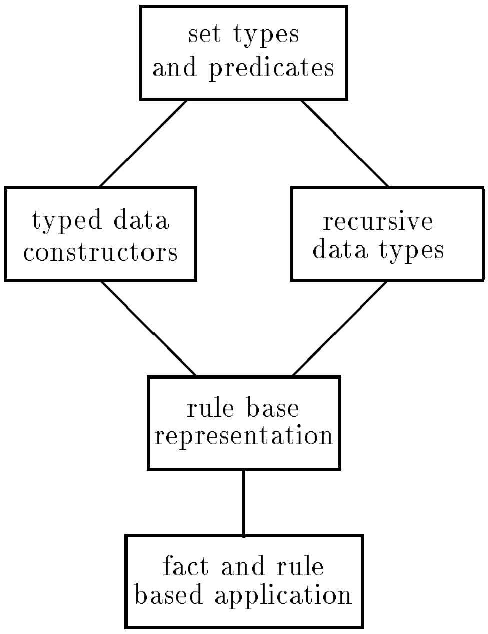

Figure 1: The Overall Structure of the Chapter

2 Fact Representation and Rule Application in Database Programs

The area of database programming languages has been an active subject of research for the past decade [BA 86]. Examples include ADAPLEX [SFL 81], Taxis [MBW 80], Plain

[WASS 79], PS-ALGOL [ACC 81], Pascal/R [SCHM 77] and DBPL [MRS 84]. Database programming languages aim at providing a clean and comfortable interface between databases and programming languages. In this section we review first of all the main concepts of the database programming language DBPL. We concentrate on the concept of rule-based data *selectors* and *constructors* and present a typed approach to rule definition and update.

2.1 Relation Types and Query Expressions

The database programming language DBPL [SM 83], [MRS 84] integrates the relational data model [CODD 70] into a strongly typed programming language. DBPL is the successor to the language PASCAL/R [SCHM 77] and uses MODULA–2 [Wirt 83] as its algorithmic kernel. For the purposes of this section, the main concept is the introduction of a data type relation. A variable of a relation type is a set of typed elements traditionally restricted to records with unstructured components (i.e. 'flat tuples'). Consider, for example, a department database consisting of library data and course data:

```
TYPE bookEntry = RECORD
                       bookId: bookIdType;
                       bookAuthors: SET OF authorType;
                       bookTitle: nameType;
                       bookTopic: ...
                       ...
                     END;

     bookRel = RELATION bookId OF bookEntry;

     courseEntry = RECORD
                        courseId: courseIdType;
                        courseName: nameType;
                        textBook: bookIdType;
                        lecturer: nameType;
                          ...
                          ...
                       END;

     courseRel: RELATION courseId OF courseEntry;

VAR Library: bookRel;
    Courses: courseRel;
```

The relation type, bookRel, is used for the declaration of the relation variable Library. An entry in the variable Library contains the bookId, the author, the title and other information. The bookId component serves as the key, i.e. it identifies a relation element uniquely. Similarly, courseId is used as the key for a relation Courses of type courseRel.
There are two types of expressions over relations: boolean and relational queries. Boolean queries are first-order predicates with existential (SOME) and universal (ALL) quantification over sorts represented by relations.
The expression

```
SOME l in Library : "Hoare" IN l.bookAuthors
```

returns the value TRUE if and only if a book authored by Hoare exists in the relation Library. Relational queries use the relational quantifier (EACH) and give access to subrelations qualified by first-order predicates.
The expression

```
{EACH l IN Library : "Hoare" IN l.bookAuthors}
```

returns the relation made up of all and only those elements of Library having "Hoare" as an element of the bookAuthors value.
The EACH-quantified-expressions are called access expressions and their semantics in DBPL are context dependent. In curled brackets, as above, they denote relations ('all-elements-at-a-time') whereas in FOR ... DO brackets they serve as iterators ('one-element-at-a-time'). A third context will be discussed in subsection 2.2 where the notion of a *selector* is introduced. The above *selective* access expressions are special cases in the sense that they select only entire elements of single relations. In their full generality, *constructive* access expressions allow the construction of relation elements that span over several relations, involving Cartesian product, projection and set union (for details see [MRS 84]), thereby providing the expressive power of relationally complete query languages. Constructive access expressions will be the basis for the notion of a *constructor* introduced in subsection 2.3.
The following example illustrates the construction and assignment of a relation that pairs the titles of books with the names of those courses that use the books as textbooks.

```
VAR CourseMaterial : RELATION OF
                     (bookTitle, courseName : nameType );

CourseMaterial := { <l.bookTitle, c.courseName> OF
                   EACH l IN Library,
                     EACH c IN Courses:
                       (c.textBook = l.bookId)} .
```

2.2 Data Selection and Rule Based Integrity Control

There are several reasons why one may not want to access or give access to an entire relation variable but only to the subrelation that fulfills a given predicate. The intention to increase data protection or integrity or to refine access granularity for more concurrency may serve as examples.
The *selector* construct introduced in [MRS 84] allows the denotation of subrelation variables from a given relation. Figure 2 shows a subrelation variable, Rel[s], selected from relation Rel by a selector s.

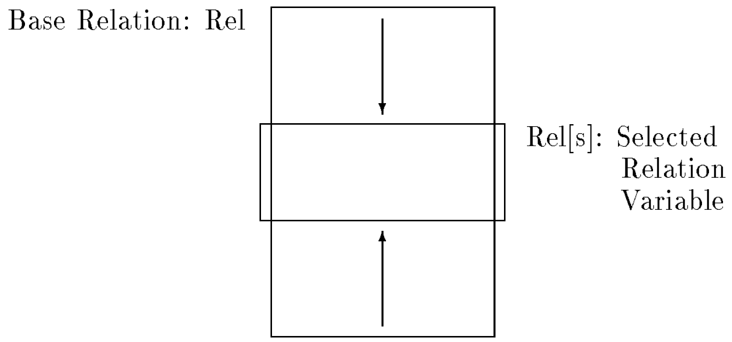

Figure 2: Selected Relations

The application of a selector to a relation variable results in a *selected variable.* In its simplest form the standard key-based selector denotes a relation element by a key value. For example,

```
Library [12345]
```

denotes the element of Library with 12345 as bookId. An update of the book 12345 can be performed by the assignment.

```
Library [12345] := newBook.
```

The semantics of this selective relation update is defined by the following assignment on the full relation variable (Compare the semantics of selective array update!).

```
IF   newBook.bookId = 12345
THEN Library := {EACH l IN Library : NOT ( l.bookId = 12345), newBook}
ELSE < exception >
```

Arbitrary first-order selection of relations can be expressed by the DBPL language construct *selector*. Formally speaking, selectors are the means to name and parameterize, i.e. to λ-abstract selective access expressions:

```
SELECTOR s FOR R: relType1 WITH (parm : parmType): relType2;
BEGIN
   EACH r IN R: p(r, parm, ...)
END s.
```

The selector s can be used to select any variable of type relType1, e.g. Rel, resulting in a selected variable, Rel[s(...)], possibly of subtype relType2. Selection is based on the selection predicate p with its formal parameters, parm, substituted by actual ones. For example, the specific key-based selector used above is equivalent to selector s if we define relType1 and relType2 to be bookRedefinee to be bookIdType, and the selective predicate p to be equal to r.bookId = parm. In other words, the denotation Library [12345] is simply shorthand for Library [s(12345)].
In general, the semantics of selected variables can be defined by giving their meaning in the two contexts they may appear: in expressions and in statements.
The value of the selected variable is the same as that of the corresponding relational expression

```
Rel[s(...)] = {EACH r IN Rel : p(r, ...)},
```

and the semantics of the selective update

```
Rel[s(...)] := rex,
```

is (essentially) equivalent to the conditional assignment

```
IF ALL x IN rex( p(x, ...) )
THEN Rel := {EACH r IN Rel: NOT p(r, ...), EACH x IN rex: TRUE}
ELSE < exception >
```

For example, the following selector when applied to a relation of type courseRel selects all elements using a specific book as textbook:

```
SELECTOR using FOR C: courseRel WITH (bookNr:bookIdType): courseRel;
BEGIN
    EACH c IN C: c.textBook = bookNr
END using.
```

This selector can be applied to a relation

```
Courses [using(12345)].
```

Selectors can be used for rule based integrity, access or concurrency control [MRS 84] as well as for the definition of access paths [JARK 85].
Formally speaking, selectors are named and parameterized (i.e. λ-abstracted) first-order access expressions. By restricting selectors to *selective* access expressions we ensure that an assignment semantics for selected relations can be given (without coping with problems like null-values or update ambiguities). In database terminology, selected variables as introduced above would be called *updatable views*.
In the following subsection we will relinquish the above restriction and admit λ-abstracted *constructive* access expressions. Roughly speaking, this will result in a trade-off between a loss in updateability and a gain in deductive power.

2.3 Rule Based Data Construction

Aside from the inability of the relational model to represent complexly structured objects adequately, traditional relational query languages have also failed to support deductive question-answering, as opposed to straight retrieval of stored facts.
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 current research [GMN 84], [KERS 84]. In [JLS 85] we introduced the notion of a *constructor* which is intended to represent a rule which derives data from stored ones. As illustrated in Figure 3, a constructor c derives a relation value {(Rel)c} from a given relation Rel.

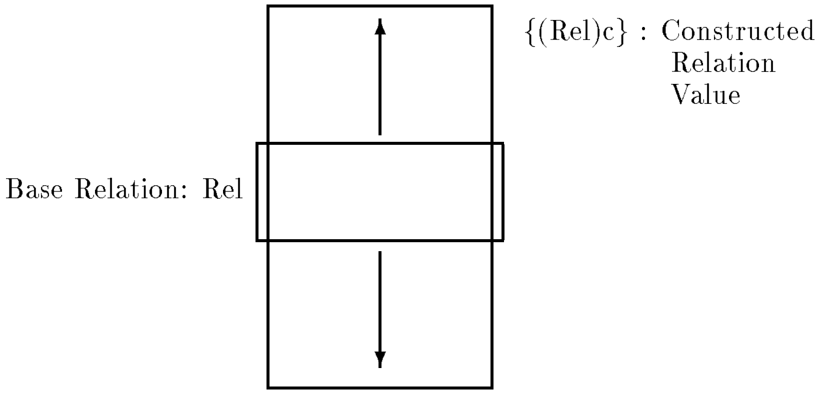

Figure 3: Constructed Relations

Assume, as an example, that the relational variable theBooks of type bookRel contains books owned and, therefore, considered important for all lecturers. Other important books are not given directly but described by rules represented by the following constructor:

```
VAR theBooks: bookRel;

CONSTRUCTOR importantFor FOR (B: bookRel)
                 WITH (personId: nameType): bookRel;
BEGIN
    EACH b IN B       : TRUE,
    EACH l IN Library : "Dijkstra" IN l.bookAuthors
         SOME c IN Courses (c.textBook = l.bookId)
                       AND (c.lecturer = personId)
END importantFor;
```

Denoting constructor application by an infix notation, the relational expression

```
{(theBooks) importantFor ("John")}
```

returns all the books from theBooks relation plus those books from the library that are either authored by Dijkstra or are used as textbooks in the courses held by John.
Constructor definitions can be recursive. As an example we introduce a relation that identifies pairs of books where one references the other:

```
TYPE pairOfReferences =
         RECORD referencing, referenced : bookIdType END;
     refPairRel =
           RELATION OF pairOfReferences;

VAR Citations: refPairRel.
```

The subsequently defined recursive constructor identifies all pairs of books such that there is a chain of references of arbitrary length from the referencing book to the referenced one.

```
CONSTRUCTOR chainedByReference FOR
       (R: refPairRel): refPairRel;
BEGIN
  EACH r IN R: TRUE,
  < r1.referencing, r2.referenced > OF
      EACH r1 IN R,
        EACH r2 IN {(R) chainedByReference}:
          r1.referenced = r2.referencing
END chainedByReference.
```

Thus, constructor application

```
{(Citations) chainedByReference},
```

denotes the relation of all pairs of referencing and referenced books that can be constructed recursively from the initial citations.
The previous example demonstrates that constructors are named and parameterized, constructive access expressions, i.e., those that may involve projection, cartesian product and union.
A naive implementation of recursive constructor application is the following loop:

```
newChain := {};
REPEAT
  oldChain := newChain;
  newChain :=
      { EACH c IN Citations: TRUE,
          <r1.referencing,r2.referenced> OF
              EACH r1 IN Citations,
                 EACH r2 IN oldChain:
                    r1.referenced = r2.referencing}
UNTIL newChain = oldChain.
```

The value of the variable, newChain, returns the least fixed point of the above relational expression [AU 79]. In [JLS 85] we give examples of mutually recursive constructors and formally define the semantics of constructors.

2.4 A Typed Approach to Rule Definition and Update

The semantics of a constructor is given by a relational query expression that specifies the data objects to be constructed from given ones. The signature of a constructor essentially determines the types of data on which the construction is based and which it returns; the body of a constructor contains the construction rule given by a parameterized constructive access expression.
Drawing the analogy to function types and function-typed variables (as provided, for example, by MODULA-2 or ADA), the set of all constructive rules (views) based on data of the same types and resulting in data of some (possibly different) type can be defined by a type definition, for example:

```
TYPE bookRule = CONSTRUCT (bookRel): bookRel;
```

Note that the set of constructive access expressions defined by the type, bookRule, is a superset of the ones given by the above constructor, importantFor, for they are all based on the same parameterized access expression given by the constructor definition. They vary only in the substitutions for their parameter, i.e., for personId. Variables of type bookRule, for example

```
VAR myImportantBooks: bookRule
```

can hold any constructive rule given by any access expression that starts from and results in data of type bookRel:

```
myImportantBooks := EACH l in Library:
                        "Dijkstra" IN l.bookAuthors.
```

Instead of defining the right-hand side access expressions from scratch, parameterized views, i.e. constructors as introduced in subsection 2.3, can be applied:

```
myImportantBooks := (theBooks) importantFor ("John");
```

Views can also be modified by redefinition,

```
myImportantBooks := EACH b IN {myImportantBooks}:
                             b.bookTopic = "database".
```

This assignment further restricts the construction of myImportantBooks to those books dealing with the subject 'database'.
As a special case we can assign the empty access expression

```
myImportantBooks :=    ;
```

which is equivalent to

```
myImportantBooks := EACH l IN Library : FALSE.
```

When evaluated

```
{myImportantBooks}
```

results in the empty relation, {}.
Many of the deeper issues of large, typed rule bases such as, for example, their structure, query languages or integrity requirements, need further substantial research. A first attempt is outlined in the subsequent section where the structures and operations of a recursive data model are used for constructor representation and management.

3 Rule Representation by Recursive Data Types

Management of large and interrelated sets of rules requires system support at least to the same extent that fact management is supported by database systems. As a first approach we discuss rule management in a database framework by mapping first-order rules into recursively defined data objects. There are several reasons why a database approach to rule management might be desirable.

First, if rules are stored as data structures, integrity can be enforced for rule updates as well as fact updates. For example, if the right-hand side of a Prolog rule is undefined, the system will simply return an empty query result (= a failing proof); the reason for the 'missing data' might have been a simple misspelling at rule definition time which could have been detected by a clever compiler.

Second, rules stored as data allow the system to return rules as query results or reactions to update requests. Situations where this feature may be useful include those in which a query predicate contradicts an integrity constraint [JARK 84], is implied by an integrity or construction rule, or returns an unexpectedly large result. In all of these cases, the user may prefer to receive a concise description of the result before deciding whether he or she wants to be answered simply by the underlying facts. For example, the query 'print courses by those professors who teach' may simply answered by 'all professors teach' if asked by a person interested in evaluating professors' performance; however, a course scheduler responsible for room assignments may want to see the actual list of all courses.

The third advantage is closely related to the second one. Experience with expert systems indicates that users tend to distrust system answers which are not simply retrieved but deduced through a lengthy chain of rule applications. The system must therefore be capable of explaining its answers. Typically, this is done by tracing the sequence of rule applications and presenting an abstract representation of the trace to the user. If rules are embedded in application programs, explanations must be programmed explicitly with each constructor. Representing construction rules as structured objects allows general explanation facilities to be implemented as search and presentation procedures on the rule base. It also facilitates temporary changes in the rules if the user believes that a rule applies currently only in modified form. For example, a company rule may require keeping three times the monthly orders as safety stock. However, the inventory manager may know that the current order level is artificially high due to a tax deadline, and may want to overwrite the rule temporarily to avoid overstocking.

In this section we introduce an extended data model based on the notion of recursive data types and we demonstrate how constructive rules as introduced above can be represented by a recursively defined rule base.

3.1 An Extended Data Model with Recursion

In the relational model, the components of each relation element are uniformly formatted data, and no component may have substructures. This limitation can become very cumbersome when the application domain requires that data objects with varying structural depth and semantic relationships are to be modelled [CODD 79]. Examples are the areas of CAD/CAM [HL 82], information retrieval [SP 82], or office information systems[GT 83]. Recursive data models try to overcome the limitations of flat relations by allowing powerful structures for relation components. Recursive data types were first introduced in the programming language

context by Hoare [HOAR 75] and applied to the data base context by [LAME 84], [LAME 85] and [LMS 84].
Lamersdorf's recursive data model provides support for the following modelling activities:

- *construction* of compound data objects from elementary ones or (recursively) from those already constructed; in the relational data model, there is only a single data type, and type construction is restricted to schema definition time;
- *selection* of specific components from compound data objects, in addition to the subset selection provided by the relational model;
- *recognition* of the construction rules or types underlying compound data objects;
- *identification* of objects by suitable object identifiers;
- *ordering* of objects.

For example, modelling a library environment by a recursive schema would allow for:

- the construction of a book instance from components as, for example, bookId, bookAuthor, bookTitle, and textual fragments;
- the selection of single book components as, for example, bookTitle or bookAuthors;
- the recognition of a given document as of type book (as opposed to article, memo, etc.);
- the identification of a document on a board of a bookshelf, and
- the ordering of words in a text.

A recursive data type is defined by a set of user-defined structure generators, used to generate instances of that recursive type. Structure generators are based on limited sets of component types which may contain other recursive data types (including the data type to be defined), or consist of simple data types as known from conventional, high-level programming languages. Components can be identified by elementary 'component selectors'. For example,

```
TYPE documentType    = ( Book  (bookId:bookIdType;
                                bookAuthors: SET OF authorType;
                                bookTitle: nameType;
                                content: textType)
                         | Article  ( ... )
                         | Memo  ( ... )
                         |       . . .     ).
```

defines a data type whose value set consists of all hierarchically structured data values which are generated by the application of the generators 'Book', 'Article', or 'Memo' to corresponding components.
Object identification can be achieved by 'mapping' object identifiers into the set of objects to be identified. Maps are sets of pairs from (Domain $\times$ Range) where 'Domain' and 'Range' are sets, and no two element pairs have the same domain value,

```
MapType   = (DomainType -> RangeType).
```

Map instances are created using a map generator

```
MapType {D1 -> R1, ..., Dn -> Rn}.
```

The operators on maps are 'DOM' (domain value set), 'RNG' (range value set), ':+'(map extention, i.e., inserting new pairs), ':&'(map update, i.e., overwriting existing pairs), ':–' (map reduction, i.e., deleting existing pairs) and 'map [argument]' (map application, i.e., identifying an object by some argument).
For example, a 'boardType' can be modelled as a map from document identifiers to the corresponding documents on a board (document identifiers are regarded as unstructured 'tokens', only subject to equality tests):

```
boardType     = (documentIdType -> documentType);
documentIdType   = TOKEN.
```

Then, for example,

```
Board :+ boardType {documentIdk -> documentk}
```

extends the content of a given board by a new document which is identified by 'documentk'.
The concepts of a relation and of a map are closely related. A relation, Rel, with keyfield, key, can be interpreted as a map of a specific type:

```
TYPE relType  = ( keyType -> elemType);
     elemType = ( ... key : keyType; ... );
VAR  Rel : relType;
```

that, wherever the mapping is defined, meets the condition

```
Rel[keyVal]. key = keyVal.
```

Another important concept supported by recursive types is that of *object ordering.* It leads to a list type data structuring mechanism:

```
listType =  LIST OF elementType.
```

Single list instances are generated by a typed standard generator for lists, listType{...}. Together with list types come the usual list operators, for example, list[i] selects the i-th element of a list.
Unordered collections of elements are represented by sets:

```
setType = SET OF elementType.
```

Sets have varying cardinality and no duplicate element values are allowed.
Considering that a book may be written by several authors, we define the component bookAuthors as

```
bookAuthors = SET OF authorType.
```

We illustrate these definitions by an example representing a bookshelf containing documents that define part of a library environment:

```
TYPE   bookShelfType   = SET OF boardType;
       boardType       = (documentIdType -> documentType);
       documentType    = (Book  (bookId:bookIdType;
                                  bookAuthors: SET OF authorType;
                                  bookTitle: nameType;
                                  content: textType)
                          | Article  ( ... )
                          | Memo  ( ... )
                          |      . . .     ) .
```

Components of Books, Articles, Memos, etc. may be defined recursively as well. For example, 'textType' may be structured by component types for paragraphs, sentences, words, etc.:

```
       textType        = LIST OF paragraphType;
       paragraphType   = ( titledPara (title: sentenceType;
                                       cont: paragraphType)
                           | UntitledPara (para: LIST OF sentenceType));
       sentenceType    =  (elems: LIST OF elementType;
                              endMark: ('.', '!', '?', ...));
       elementType     = ( Word (wordType)
                           | Mark (',', ';', ...));
       wordType        = LIST OF CHAR.
```

Finally, we are able to declare a single compound variable representing a bookshelf with all its structured components by:

```
       VAR bookShelf: bookShelfType.
```

3.2 A Recursive Representation of Rule Bases

Hiding rules within application programs results in disadvantages similar to those of hiding facts. Therefore, many of the historic arguments that led to the development of databases also hold for rule bases. Rules concentrated in a rule base can be more easily altered according to the needs of changing application requirements, and there are better means of controlling the integrity of interrelated rule sets. In addition, a rule base can be shared by a user community, and valuable system support such as optimization, recovery and distribution can be provided. Subsequently we define the schema of a generic rule base together with a set of elementary procedures for rule base management. Our rule base definition makes heavy use of the recursive data model introduced in the previous subsection.

```
    TYPE ruleSchema = (nameType -> ruleTypeExtension);
         ruleTypeExtension = RTypeExt(ruleType: ruleSignature;
                                      ruleExt: nameType -> ruleExpression);
         ruleSignature = RType (forTypes: LIST OF nameType;
                                     resultType: nameType);
         ruleExpression = LIST OF relationElement.
```

The data type, ruleSchema, defines a mapping between names and pairs of rule type definitions and their extensions. The elements of the extension are named too and represent DBPL access expressions as introduced in section 2. The remaining recursive types given below model those access expressions and can be derived mechanically from the DBPL syntax definition [EEKM 85].

```
relationElement = (  Expr(e: expression)
                   | Selec(s: selection)
                   | CompSelec(c: componentSelection) );

selection = Subrelation( eachList: LIST OF elementDenotation;
                         predicate: expression );

elementDenotation = EachIn( variableName:nameType;
                            expr: expression );

componentSelection = Of( aggr:aggregate;
                         Subrelation:selection );

expression = (  QuantifiedExpr (
                   quantifier  : ("SOME", "ALL");
                   variableName: nameType;
                   inExpr      : expression;
                   predicate   : expression )
              | Operation(left : expression;
                         op    : operator;
                         right : expression  )
              | Factor(f: factor)
              |    .
                   .
                   . );

parmType  = ... ;

aggregate = ... ;
nameType  = ... ;
factor    = ... ;
operator  = ... .
```

Next we define a set of procedures that work on a variable ruleBase of type ruleSchema and introduce and manipulate rule types and rule extensions. We make use of predefined procedures, MakeT, that accept parameters of type string and return corresponding objects of type T.

The first procedure creates a named and empty rule extension set that is able to hold rules of some rule type given by its signature, ruleSign:

```
PROCEDURE createRuleTypeExt (VAR ruleBase: ruleSchema;
```

```
                                    ruleExtName: nameType;
                                      ruleSign: string);
  BEGIN
    IF ruleExtName IN DOM ruleBase
    THEN (* exception: name already in use *)
    ELSE ruleBase :+
            {ruleExtName -> RTypeExt(makeruleSignature (ruleSign), {})}
END createRuleTypeExt;
```

The rule type, bookRule, defined in subsection 2.4 can be introduced into our rule base by the following statement:

```
createRuleTypeExt (libRuleBase,
                    bookRules,
                 "CONSTRUCT (bookRel): bookRel").
```

Note that the set of rules associated with the extension, bookRules, is still empty. The following procedure creates a named and typed rule instance:

```
PROCEDURE createRuleInstance (VAR ruleBase: ruleSchema;
                              ruleExtName, ruleName: nameType);
  BEGIN
    IF ruleExtName IN DOM ruleBase
    THEN WITH ruleBase[rulExtName] DO
      IF ruleName NOT IN DOM ruleExt
      THEN ruleExt :+ {ruleName -> {}}
      ELSE (* exception: rule name already in use *)
    ELSE (* exception: unknown rule set *)
    END
END createRuleInstance.
```

For example, the empty rule named myImportantBooks is created by:

```
createRuleInstance (libRuleBase, bookRules, myImportantBooks).
```

Assignment of arbitrary rule expressions to rule variables within some rule extension is accomplished by the

```
PROCEDURE assignRuleInstance (VAR ruleBase: ruleSchema;
                                ruleExtName, ruleName: nameType;
                                    ruleExpr: string);
  VAR ruleObject: ruleExpression;
  BEGIN
    ruleObject := makeRuleExpression(ruleExpr);
    IF ruleExtName IN DOM ruleBase
    THEN WITH ruleBase[ruleExtName] DO
      IF ruleName IN DOM ruleExt
      THEN IF typeCompatible(ruleObject, ruleType)
```

```
            THEN ruleExt :& {ruleName -> ruleObject}
            ELSE (* exception: rule not type compatible *)
       ELSE (* exception: unknown rule name *)
     ELSE (* exception: unknown rule set *)
     END;
END assignRuleInstance.
```

The type checking procedure, typeCompatible, works on its two recursive parameters of type ruleSignature and ruleExpression and will return any error messages that would we expect from a good compiler.
The first example of a rule assignment in subsection 2.4 now reads

```
assignRuleInstance (libRuleBase,
                    bookRules, myImportantBooks,
                    "EACH l IN Library: 'Dijkstra' IN l.bookAuthors").
```

These procedures sketch only the most elementary interface to a rule base. In the next section they are used to outline a set of higher level functions for rule base management.

4 Rule Bases: Application to Integrated Fact and Rule Management

In this section we apply the elementary procedures defined above to sketch some of the functionality of an integrated fact and rule management system.

4.1 Example 1: An Interactive Rule Base Management System

The first example is a simple application of the above procedures for rule base management. We provide a naive interactive interface which allows the introduction of library users and the creation and alteration of one set of rules per user for the construction of the books he considers important. In addition to the rule base, our dialogue interface rule base is supposed to know at least the following two relations of type bookRel: theBooks refers to those books considered important for all users, Library contains all the books known within our application.

```
VAR theBooks,
    Library     : bookRel;
    libRuleBase : ruleSchema;
    Command     : STRING;
    userName,
    userRuleName: nameType;
    userRuleExpr: STRING;

BEGIN

  libRuleBase := {};
  read(Command);
  WHILE Command <> "STOP" DO
    CASE Command OF
      "NewUser":
          read(userName);
          createRuleTypeExt (libRuleBase, userName,
                              'CONSTRUCT (bookRel): bookRel');
          createRuleInstance (libRuleBase, userName, Obligatory);
          assignRuleInstance (libRuleBase, userName, Obligatory,
                                       "EACH b IN theBooks: TRUE")
     | "DeleteUser"
          read(userName);
          deleteRuleTypeExt(libRuleBase, userName)
     | "NewUserRule"
          read(userName);
          read(userRuleName);
          createRuleInstance(libRuleBase, userName, userRuleName)
     | "DeleteUserRule" ...
     | "AssignUserRule"
```

```
          read(userName);
          read(userRuleName);
          read(userRuleExpr);
          assignRuleInstance(libRuleBase, userName,
                             userRuleName, userRuleExpr);
    | "ReturnByRule"
          read(userName);
          read(userRuleName);
          outputRel({libRuleBase[userName].ruleExt[userRuleName]});
    | "ReturnAll"
          read(userName);
          WITH libRuleBase[userName] DO
            FOR EACH ruleName IN DOM ruleExt DO
              outputRel({ruleExt[ruleName]});
    END
  END
END.
```

The command NewUser creates for each user a set of rules (identified by the user's name) and initializes it by one rule named Obligatory. This rule selects all the entries of the relation theBooks. The procedure ReturnByRule prints the books constructed for a named user by a named rule and ReturnAll shows all the books considered important by a named user.
We now provide an example dialog through which users evaluate and alter our rule base through the above interactive interface.

```
NewUser John
NewUser Jim
NewUser Peter

NewUserRule John
            BooksByHoare

NewUserRule John
            BooksForCourse101

AssignUserRule John
               BooksByHoare
               "EACH l IN Library: 'Hoare' IN l.bookAuthors"

AssignUserRule John
               BooksForCourse101
               "EACH l IN Library: SOME c IN Courses
                                   (c.courseId = 101) AND
                                   (c.textBook = l.bookId)"
```

In this state, the rule extension for John contains the following named rules

```
{Obligatory  -> EACH b IN theBooks: TRUE;
 BooksByHoare -> EACH l IN Library: "Hoare" IN l.bookAuthors;
 BooksForCourse101 -> EACH l IN Library: SOME c IN Courses
                          (c.courseId = 101) AND
                          (c.textBook = l.bookId)}
```

The corresponding representation of the rule base, libRuleBase, can be depicted by the following data object of the recursive type, ruleSchema:

```
{ John ->
        RTypeExt(RType ({bookRel}, bookRel),
                { Obligatory ->
                    {Selec(
                       Subrelation(
                         EachIn (b, Factor(theBooks)),
                         Factor(TRUE)
                                  )
                          )
                    }
                  BooksByHoare ->
                    {Selec(
                       Subrelation(
                         EachIn(r,Factor(Library)),
                         Operation (
                           Factor(Hoare)
                             IN
                           Factor(r.bookAuthors),
                                   )
                                  ),
                    }
                  BooksForCourse101 ->
                    {Selec(
                       Subrelation(
                         EachIn(r,Factor(Library)),
                         QuantifiedExpr(
                           SOME, c, Factor(Courses),
                           Operation     (
                             Operation(Factor(c.courseId),=,Factor(101)),
                               AND
                             Operation(Factor(c.textBook),=,Factor(r.bookId))
                                         )
                                       )
                                  )
                          )
                    }
      },
  Jim   -> ...
```

```
    Peter -> ...
  } .
```

We can delete user-defined rules, for example,

```
DeleteUserRule John
               BooksForCourse101
```

and create new rules based on existing ones:

```
NewUserRule John
            BooksByDijkstraAndHoare

AssignUserRule John
               BooksByDijkstraAndHoare
               "EACH l IN {BooksByHoare}: 'Dijkstra' IN l.bookAuthors"
```

Now the rule extension for John looks like

```
  {Obligatory  -> EACH b IN theBooks: TRUE;
   BooksByHoare -> EACH l IN Library: 'Hoare' IN l.bookAuthors;
   BooksByDijkstraAndHoare -> EACH l In {EACH l IN Library:
                                            "Hoare" IN l.bookAuthors}:
                                            "Dijkstra" IN l.bookAuthors  }
```

Rule application that leads to database querying is illustrated by

```
ReturnByRule John
             BooksByDijkstraAndHoare
```

(* prints all books in the library co-authored by both, Dijkstra and Hoare *)

```
ReturnAll Jim
```

(* prints the entries in theBooks relation *)

4.2 Example 2: Integrity Management and Explanation Component

The above rule base schema maintains only those integrity rules that are inherent in the underlying recursive data model. Uniqueness of names, for example, is enforced for rule sets by the fact that the type, ruleSchema, is based on a map definition, and that maps disallow duplicates in their domains. However, as with databases, we would also like to specify a rich set of application dependent constraints on rule bases. For example, we might want to enforce that the number of rules within a rule set does not exceed, say, 20, or that the name used to identify a rule set is different from the names that identify individual rules within that set. The subsequent examples show how the notion of a selector, as discussed in subsection 2.2 in the context of database integrity, can also be utilized to define constraints on rule bases.
Our first example selector restricts the components of a variable of type ruleSchema, i.e. , the individual entries of the underlying mapping, to those where the number of rules in the rule extension is less than or equal to n:

```
SELECTOR s1 FOR ruleBase: ruleSchema WITH (n: CARDINAL): ...;
BEGIN EACH r IN ruleBase: count ((RNG r).ruleExt) <= n) END s1;
```

The selector semantics allow only those assignments to the selected rule base, libRuleBase[s1(20)], that maintain the intended selection predicate.
Our second example shows how the integrity constraint to differentiate names for rule sets and for rules within rule sets can be enforced:

```
SELECTOR s2 FOR ruleBase: ruleSchema ...;
BEGIN EACH r IN ruleBase: NOT (DOM r IN DOM ((RNG r).ruleExt)) END s2.
```

As indicated in section 4.1, query processing in the integrated fact and rule management system will intermix rule retrieval and data retrieval, i.e., access to the recursive data structures of the rule base and to the relations of the data base. For example, if a user asks for all the books important for John, the query processor will first submit a sequence of queries to the rule base to retrieve the selected rule definitions. In the example (see subsection 4.1), the rules for John define the following set of data base queries:

```
{ EACH b IN theBooks: TRUE},
{ EACH l IN Library: 'Hoare' IN l.bookAuthors},
{ EACH l IN Library: { 'Hoare' IN l.bookAuthors}:
                       'Dijkstra' IN l.bookAuthors}.
```

The query processor assembles an internal representation of the above database query set based on the syntax-tree like answer to the rule base query. The resulting database query may be optimized before it is evaluated; the nested third subquery in the example above can be transformed into the 'flat' query

```
{ EACH l IN Library: {"Dijkstra", "Hoare"} IN l.bookAuthors}
```

(unless there is an index on the individual authors that may make the evaluation of the nested form more efficient [KOCH 85] [JK 83]). The further processing of recursive queries has been discussed elsewhere (compare, eg., the survey paper by [BR 86] or the chapter by Gardarin et al.). Here we are concerned with the utilization of the retrieved rule set for explaining query results.
If a user asks for a general explanation of a query result ('why these data?'), the system can simply display a suitable representation of the rule set involved. If mnemonic rule names have been selected, a name list may provide a simple but effective explanation. For John's important books this would be

```
Obligatory
BooksByHoare
BooksByDijkstraAndHoare
```

The user may then request further details by asking for an explanation of one of these:

```
Obligatory ?
```

whereupon the system returns a representation of the expression:

```
EACH b IN theBooks: TRUE .
```

Alternatively, the user could also doubt the relevance of a particular subset of an answer. The system should then determine the specific 'rule set of support' for this subset, i.e., the rule application leading to this particular result. The user may use this information to modify the particular rule. For example, if all the books in the library authored by Hoare are obligatory anyway, i.e., they are also elements of the relation theBooks, the user may decide to strengthen the rule that determines his favorite programming language books and instead query the library for books collectively authored by Dijkstra and Hoare (compare examples in subsection 4.1).

5 Concluding Remarks

Some of the recent extensions to set- and predicate-oriented data models have attempted to model complex objects, while others tried to increase the power of query languages for conventional relational data. This chapter presents the case that the combination of recursive query languages with recursive object types is an effective strategy for the integrated management of large fact and rule bases. We have demonstrated the potential of this approach and outlined some implementation techniques based on relational technology. However, further substantial research will be required for a closer integration of the two approaches, focusing in particular on database models where the facts are also complex objects.

6 References

[ABCE 76]

Astrahan,M.M.; Blasgen,M.W.; Chamberlin,D.D.; Eswaran,K.P.; Gray,J.N.; Griffiths,P.P.; King,W.F.; Lorie,R.A.; McJones,P.R.; Mehl,J.W.; Putzolu,G.R.; Traiger,I.L.; Wade,B.W.; Watson,V.: System R: Relational Approach to Database Management, ACM Trans. on Database Systems, Vol.1, No.2, (June 1976), 97-137

[ACC 81]

Atkinson,M.P.; Chrisholm,K.; Cockshott,P.: The New Edinburgh Persistent Algorithmic Language, University of Edinburgh, Department of Computer Science, CSR-90-81, August 1981

[AU 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

[BA 86]

Buneman,P.; Atkinson M.: Inheritance and Persistence in Database Programming Languages, Proc. ACM SIGMOD Conf., Washington,D.C., May 28-30, 1986, 4-15

[BL 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

[BMS 84]

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

[BR 86]

Bancilton,F.; Ramakrishan,R.:An Amateur's Introduction to Recursive Query Processing Strategies, ACM SIGMOD Conf., Washington,D.C., May 1986, 16-52.

[CH 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

[CM 81]

Clocksin,W.F.; Mellish,C.S.: Programming in PROLOG, Springer Verlag 1981

[CODD 70]

Codd,E.F.: A Relational Model of Data for Large Shared Databanks, Comm. ACM, Vol.13, No.6, June 1970, 377-387

[CODD 79]

Codd,E.F.: Extending the Relational Database Model to Capture More Meaning. ACM TODS, Vol. 4, No.4, December 1979

[EEKM 85]

Eckhardt,H.; Edelmann,J.; Koch,J.; Mall,M.; Schmidt,J.W.: Draft Report on the Database Programming Language DBPL, DBPL-Memo 091-85, Johann Wolfgang Goethe-University Frankfurt, Frankfurt 1985

[GMN 84]

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

[GT 83]
Gibbs,S.; Tsichritzis,D.: A Data Modelling Approach for Office Information Systems, ACM Transactions on Office Information Systems, Vol.1, No.4, October 1983

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

[HOAR 75]
Hoare,C.A.R.: Recursive Data Structures. Intern. Journal of Computer and Information Science, Vol.4, No. 2, 1975

[ISW 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]

[JARK 85]
Jarke,M.: Common Subexpression Isolation in Multiple Query Optimization, **in** Kim,W; Reiner,D.S.; Batory,D.S. (eds.): Query Processing in Database Systems, Topics in Information Systems, Springer Verlag 1985

[JK 84]
Jarke,M.; Koch,J.: Range Nesting: A Fast Method to Evaluate Quantified Queries, Proc. ACM-SIGMOD Conf., San Jose, May 1983, pp. 196-206

[JLS 85]
Jarke,M.; Linnemann,V.; Schmidt,J.W.: Data Constructors: On the Integration of Rules and Relations, 11th Int. Conference on Very Large Data Bases, Stockholm, August 1985

[JV 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

[KHI 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

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

[KOCH 85]
Koch J.: Relationale Anfragen Zerlegung und Optimierung, Ph.D dissertation (in German) 1985, Informatik Fachberichte Vol. 101, Springer Verlag

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

[LAME 85]
Lamersdorf,W.: Semantische Repräsentation komplexer Objektstrukturen: Modelle für nichtkonventionelle Datenbankanwendungen, Ph.D dissertation (in German) 1985, Informatik Fachberichte Vol. 100, Springer Verlag

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

[MBW 80]
Mylopoulos,J.; Bernstein,P.; Wong,H.T.K.: A Language Facility for Designing Interactive Database-Intensive Applications, ACM TODS, Vol.5, No.2, June 1980

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

[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

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

[SL 85]
Schmidt,J.W.; Linnemann,V.: Higher Level Relational Objects, Proc. Fourth British National Conference on Databases (BNCOD 4), Cambridge University Press, July 1985, 1-23

[SM 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

[SMIT 84]
Smith,J.M.: Expert Database Systems: A Perspective, **in** [KERS 84]

[SP 82]
Schek,H.J.; Pistor,P.: Data Structures for an Integrated Data Base Management and Information Retrieval System. Proc. 8th Int. Conf. on VLDB, Mexico City, September 1982, 197-207

[WASS 79]
Wasserman,A.I.: The Data Management Facilities of PLAIN. Proc. ACM SIGMOD Conf., Boston, May 1979

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