



Instituut voor Taal, Logica en Informatie

Faculteit der Wiskunde en Informatica, Roetersstraat 15, 1018WB Amsterdam, or
Faculteit der Wijsbegeerte, Grimburgwal 10, 1012GA Amsterdam

**A SEMANTICAL MODEL FOR INTEGRATION
AND MODULARIZATION OF RULES**

Peter van Emde Boas
Department of Computer Science
University of Amsterdam

Received October 1986



Institute for Language, Logic and Information

Faculty of Mathematics and Computer Science, Roetersstraat 15, 1018WB Amsterdam, or
Faculty of Philosophy, Grimburgwal 10, 1012GA Amsterdam

Abstract : In the formalization of rules of administrative, legislative or fiscal nature three different conceptual frameworks come into mind, all of which can be recognized as representing relations. Rules which list a finite collection of cases can be encoded using the tables from the relational database model. Other rules have the structure of logical implications; these can frequently be formalized in the format of Horn clauses, establishing a link to a subset of general Logic Programming for which a relational semantics is known. Finally there are algebraic identities and inequalities which again lead to a relational semantics if considered from the perspective of Algebraic Geometry. Complementary to these three conceptual frameworks are three current technologies for storing and retrieving information: relational databases, production rule based expert systems and spreadsheets.

We propose a family of languages, called **RL**, which unifies these three frameworks into a single relational semantical model for modules consisting of rules which come in three flavours: tabular rules, clauses, and constraints. We give an outline for a syntax and semantics. We also discuss the issue of modularization of large collections of rules.

May 1986

Keywords : Relational database, Horn Clauses, Constraints, Regulations

Math. Subject Classes : 68Q55, 68P15, 68T30

Comp. Reviews Class : F 3.2, H 2.3, I 2.4, J 1

This paper is the author's contribution to the proceedings of MFCS'86 to be held in Bratislava, Aug 1986.

1. Introduction

In this paper we attempt to give an answer to the following question: *What does it mean to store the information expressed by a hierarchically structured collection of Rules in a system?* By the word *Rules* we have in mind some formalized representation of a collection of regulations of administrative, legal or fiscal nature. The phrase *hierarchically structured* reflects the fact that collections of rules are organized in chapters, sections and articles, where each fragment has some relevance independent of the context. For example the rules of taxation can be specialized into an income tax regulation or a sales tax regulation, which in its turn can be specialized for use in different countries. *Storing these rules in a system* means: building something on top of which a real life administrative system can be designed in such a way that it can use the rules as another piece of accessible information.

The situation can be compared to the role of a Database system in an administrative environment. The database stores the factual information on, say the employees of your firm, whereas the application program contains the information needed to know how to process these data in order to produce paycheques. If there were no database the application program would have to obtain the data from some file, or these data would have to be listed explicitly in the program itself. Both cases lead to update problems when the information stored in the data changes. Databases are designed to be a reasonably safe environment where (among others) such updates can be performed. They provide the application programmer with a tool separating the fluctuating data from the more stable structures in the system. They provide the additional feature that data can be shared between several application programs and/or users.

The rules system we have in mind is intended to play a similar role. In real life (administrative) rules themselves are fluctuating also. Moreover, the same body of administrative rules can be required for several application programs. The very same arguments for the separation between application program and data we saw above can be invoked in support for a *separation between rules and application programs*. After the factual data the rules themselves become something which should be factored out of the application program and which should be stored in a separate repository, called a *Rules base*.

Currently in real life application programs part of the rules are implemented by the application program and part of it can be found inside the database. Ideally, an enhanced database might store all the information contained in the rules, but this is beyond the power of current technology. This is caused by the fact that databases are designed to cope only with finite chunks of information in tables. These tables subsequently can be subjected to operations from an extended relational algebra, yielding new tables derived from the stored tables. All tables remain finite. But in reality we encounter a much larger variety of conceptual frameworks for expressing regulations. Some rules are indeed expressible by listing a finite table of cases, like the relation between a county and its local sales tax rate. But the rule that the sales tax is obtained by

computing that particular percentage of the price of the item sold, and has to be added to this price in order to produce the price of the item can be expressed by an algebraic identity. Finally, a rule expressing that, if an item X is exempt for sales tax then its spare parts are exempt as well (provided they are sold in a single order), can more conveniently be expressed in some logical rule written in a language derived from first order predicate logic. So next to pure relations we would like to structure our rules using the expressive mechanisms of *clauses* and *constraints*.

Corresponding to these alternative frameworks for expressing rules we have contemporary available systems for storing and accessing such information. Finite tabular information is the topic for database systems. Spreadsheets are widely used in the context of algebraic identities, and logical rules are the basis of expert systems and logic programming projects in general.

With these three separate systems in mind we now can propose a more specific formulation of our initial question. We are aiming for a *Rules technology* which deals with an enhanced database incorporating features from spreadsheet technology and of logic programming. The user should be able to express his knowledge about the rules using relational algebra, algebraic constraints and logical rules together. The representation of the rules should be structured according to a similar hierarchy as the real life rules. The representation of the rules should also be *auditable* in the sense that the written text can be inspected by a non-technician in order to convince himself that the rules in the system indeed represent those in the world outside. The semantics of such a collection of rules should be represented in a relational framework. Having stored this information the user should be able to *query* this information like any other database.

In the present paper I will restrict myself to the semantical base for such a technology. We present the outlines of a language **RL** for describing rules together with the relational model for interpreting this language. Next we have the problem of *modularization*. How do we accommodate for the fact that rules can be grouped together in chapters and sections, each having a local meaning, which may depend on each other for being interpreted. This problem has been dealt with in my report [V85] on which this paper is based, but due to the size of the subject I must restrict myself to some side remarks on this issue in the present paper.

The language design and model described in this paper was developed during an eight month visit to the IBM San Jose Research Center, where I was connected to the *Rules Technology project*, chaired by Peter Lucas. This project aimed at a new way of programming large scale applications dealing with real life fiscal and organizational information. The essence of the new technology is the separation between the factual information describing the state of affairs dealt with from the algorithmic part of the application. The rules should be stored in a shared repository called *rules base*, which should be accessible using multiple interfaces, varying between SQL and ordinary programming languages like Pascal or PL/1. See [Lu80,Lu85] for more motivation. The results of my participation are available in the internal IBM report [V85]. The present paper represents an attempt to extract the semantical ideas from this rather overloaded and inaccessible document.

One final remark is in order before I proceed to the technical part of this paper. The language **RL** introduced in this paper is not a single well defined language. It represents rather a family of **RL**-languages, since at several places I have abstained from making preemptive decisions on what should be in the language and what should not be there. At the present level of generality it does not make sense to propose a definite outline of the specific Abstract data typing mechanism used, the available type constructors or the extent of the relational algebra available in the language. Least of all I want to stipulate a specific syntax; any resemblance of the proposed (surface) syntax to that of any known programming language is entirely coincidental and caused, if caused by anything at all, by the education of the author and the text editing features on the Macintosh system on which this paper was prepared. What these languages should share is their common conceptual structure and semantic model for assigning meanings to rules, and that is the subject of this paper.

2. Relational framework.

We assume that the reader is familiar with the framework of relational databases (e.g. [U82]). The remarks in this section are intended to illustrate my perspective on this framework, which is primarily inspired by that of Imielinski & Lipski [IL84].

Domains are sets which can be equipped with operations. There are primitive domains like integers, reals, characters and character strings, and user defined domains which can be obtained using the standard tools from contemporary programming languages, including abstract data types. Domains can both be finite or infinite.

There are good reasons for including an abstract data typing mechanism in **RL**. For example, within administrative applications dates are a relevant data type, and calendar arithmetic represents an important part of the computations performed. It is a typical task which is a candidate for being factored out of the rest of the programming job, both for modularization and auditability. Current programming languages use abstract data types as tool for achieving this separation. So why not turn dates into an abstract data type?

Relations are subsets of Cartesian products of domains. We share with the relational database model the fact that our products are formed using domains tagged by an identifier called the *attribute*. Factors of a product should have different attributes. Although attributes in a single relation are required to be distinct, two different relations can share one or more attributes, but in this case the corresponding domains must be equal. Attributes provide access to coordinates of tuples in the product or in the relations. A tuple can be considered to be a function from the attributes into their corresponding domains, and the relation becomes a collection of such tuples. Alternative names for relations and tuples are *Tables* and *Rows*. The *type* of a relation consists of set of pairs formed by the attributes and their corresponding domain types.

There exists a variety of operations which can be performed on relations leading to the subject of *relational algebra*. Some of these operations like union and intersection, are inherited from the language of set theory. The additional twist is to define the union or intersection of two relations in the situation where these relations are subsets of two different Cartesian products with different sets of attributes. In this situation the argument relations are first extended to relations defined over the union of the two attribute sets by forming the product with the entire domains corresponding to the absent attributes. Next the set theoretical operations union and intersection have their ordinary meaning.

The intersection operation, under this interpretation becomes an extremely powerful operation which includes now the standard intersection, the Cartesian product of two relations, and the operation known as the *Natural join*. These cases are obtained if the two attribute sets are equal, disjoint or arbitrary, respectively. For more on this perspective on relational algebra and its connection with Cylindric Algebra see [IL84].

Another important operation of relational algebra is the projection of a relation on a subset of its attributes. If we consider the relation to be a set of functions on the attributes the projection forms the set of restrictions of these functions to the subset of attributes. Multiple rows are identified into a single row.

Flexibility with respect to the identity of attributes is introduced in **RL** by introducing some *renaming* mechanism. This enables the construction of isomorphic copies of a given relation where only the attributes have become new identifiers. This renaming mechanism makes it possible to form the natural join of a relation with itself in a non-trivial way. Projection and renaming are combined into a single *present* operation.

Another type of operation which does not belong to the relational algebra proper, but which is available in some relational database systems is the *Aggregate*. In this type of operation a relation is grouped together according to the values of some attributes, while at the same time by use of some arithmetical operation a set of values of another attribute is merged into a single value. A typical example is the operation which groups shipping orders by customer in order to compute their total amounts due by summation over the price attribute. In stead of summation other operations like counting, maximizing or averaging can be used.

3. Tabular rules, clauses and constraints.

The basic ideas behind the **RL** language and model are expressed by the slogan: *Everything is a relation, but not all relations are equal*. All rules will be interpreted by relations. The word relation refers to relations in the framework explained in the previous section. The inequality mentioned in the above slogan does not refer to the relations themselves as a set of tuples of objects, but to the linguistic and conceptual construction which is used for defining the relation in

RL and to the way relations are combined in order to obtain their joint meaning. It is precisely this inequality which enables us to build our unified framework for relational database expressions, algebraic equations and inequalities and logical rules.

The present section will introduce the three sorts of rules available in **RL** : *Tabular rules*, *Clauses* and *Constraints*. Syntactically they will be different categories of rules. Their contribution to the meaning of a program is also different. Tabular relations and clauses will define named relations, but where each tabular rule determines a *tabular relation* by itself, the clauses together must be evaluated as a system in order to obtain the collection of *clausal relations* which represents their meaning; this system moreover may be a recursive system. The constraints together determine a single relation called the *principal relation*. Since this relation is not defined by a definition which occurs in the program it becomes an *unnamed relation*. There is also a difference concerning adding rules to the program. If a tabular relation has been defined no further rule for this relation can be added. For a clausal relation a new rule can be added and in this case the relation may become larger; the new clause introduces a new alternative. On the other hand, if we add a new constraint the principal relation will become smaller.

For each type of rule we first give some intuition on their conceptual origin and intended meaning, followed by their syntax and a more formalized semantics. For abbreviation we use in the syntax the following rules schemata (hyperrules in the style of [vW75]):

```

<Xoption> ::= <X> | <empty>
<Xsequence> ::= <X> | <X><Xsequence>
<Xlist> ::= <X> | <X> , <Xlist>
<Xname> ::= <identifier>

```

So whenever *X* is replaced by the name of some syntactic construct, say the construct term, the above four hyperrules tell you that a *termoption* is either a term or the empty string, that a *termsequence* is either a term or a term followed by a *termsequence*, etc. .

A complete **RL** program will consist of a heading followed by a sequence of rules. In the heading one finds among other information a list of attribute names together with their domain types, so within the rules themselves a relation type can be specified by listing a set of attribute names.

3.1 Tabular rules.

In the relational database model relations either are primitive (stored in the database as a physical table) or derived by means of operations in the relational algebra (view definitions) or an extension thereof like the formation of aggregates. Derived relations are not created physically in

the database and stored, unless the user in querying the database explicitly asks for this to happen. Derived relations become intensional objects.

What these relational constructs share is that they provide a single definition for a relation. It is not allowed to extend the definition of a derived relation by writing down another part of its definition in another place of the program. Their meaning is determined once for all. Their meaning changes only if the physical tables referenced in their definition are subjected to updates but also then the dependency between the derived relation and the database tables remains the same.

In **RL** we will introduce a category of constructions of this nature which will be called *Tabular Rules*. They will be interpreted as definitions of relations. The relations defined by Tabular rules are called *Tabular Relations*. The following four types of tabular rules can be found in **RL** :

- explicit external reference
- explicit listing
- local view definitions
- aggregates

Explicit external references are used for importing a database table from a connected database in an **RL** program. An Explicit listing includes a table written in textual form. These syntactic structures have mainly as purpose the specification of name, domain types and attribute names under which these explicit relations are known inside the program. Local view definitions are the equivalent of ordinary view definitions in a database. Aggregates are definitions of relations formed from existing ones by grouping according to some attributes and performing arithmetic like summation on others. For the last two constructs it is required that all relations used within a definition have been defined by a tabular rule occurring earlier in the program. This means that recursion is not allowed for tabular rules. For local view definitions and aggregates only a name has to be specified in the program; attribute names and domain types are determined implicitly by the syntax of the construct.

3.1.1 Syntax of tabular rules.

```
<tabular_rule> ::= <relation_typing> = <table_definition> | <relationname> == <construct>
<relation_typing> ::= <relationname> ( <attributenamelist> )
<table_definition> ::= <explicit_external_reference> | <explicit_table_listing>
<construct> ::= <local_view_definition> | <aggregate>
<local_view_definition> ::= <relational_algebraic_expression>
```

```

<aggregate> ::= aggr <relationname> by <attributename> over <attributename>
               to <attributename> <orderingoption> under <aggregating_operator> eaggr
<ordering> ::= order <ordering_operator> on <attribute_name>

```

Some examples of tabular rules:

```

sales(customer, object, price) = external table X12.SALES
vat_rates(class, percentage) = ((exempt,0),(low,6),(high,20))
salesinfo == proj( category(object,class) join vat_rates(class, percentage) join
               sales(customer, object, price) )[customer,percentage,price]
totals == aggr sales by customer over price to totalprice under + eaggr

```

Tabular rules are definitions of the form $X = Y$ or $X == Y$; in the first case the identity of the attributes is explicitly provided inside X , whereas in the second case this information can be synthesised from the construct Y . We leave the specific syntax for external references and explicit listings unconsidered. Also the specific syntax for expressions in the relational algebra which can occur inside view definitions is left unspecified. For the aggregates we have proposed a syntax mainly in order to indicate the relevant fields of an aggregate definition; this definition should specify:

- the relation from which the aggregate is constructed ; aggr-field
- the attributes on which this relation is grouped ; by-field
- the attribute subjected to arithmetic merging ; over-field
- the attribute-name which obtains as value the result of this arithmetical merging ; to-field
- the arithmetical operation invoked in the merging ; under-field

Since we do not require that the operation in the under-field is either associative or commutative the order in which the arithmetical merging is performed may be relevant and therefore the programmer may need to specify an order. This specification consists of an ordering relation in the order-field, and an attribute-name in the on-field to specify the attribute on whose values the ordering is determined. The idea of including such a general aggregating mechanism was inspired by the "generalized quantifier approach" in [dB84].

3.1.2 Semantics of tabular rules.

A tabular rule defines a single named relation. The name of this relation is the relation name in the left hand side of the rule. For explicit external references and listings attributes are included in the relation typing. For the case of an external reference the right hand side must provide the

necessary information for importing a conforming table from the database. In the case of an explicit listing the right hand side is interpreted as a table of constants; these constants are evaluated and grouped into a table conforming to the attributes and types given in the relation typing.

Both in the case of a local view definitions and an aggregate the result is a relation whose attributes and types are fully determined by the right hand side of the rule. For a local view definition the right hand side expression is evaluated in the relational algebra. For the case of an aggregate the relation in the aggr-slot is considered; this relation is projected on the attributes occurring in the by-slot, the over-slot and the order-slot. For each tuple of values for the attributes in the by-slot which occurs in this relation the following action is performed: first the set of rows extending this tuple is formed, and ordered according to the operation specified on the attribute in the order-slot. Next the sequence of values of the attribute in the over-slot are combined into a single value by application of the aggregating operation specified. The resulting value becomes value for the attribute in the to-slot and the attributes in the over-slot and the order-slot are projected away. Duplicate rows in the result are replaced by a single copy.

3.2 Clauses.

The second sort of rules we have in mind are the *Clauses*. These clauses form a rather restricted set of Horn clauses which are included in **RL** in order to make available some nice features of Logic Programming. We aim for including exactly that part of logic programming which has a nice relational semantics.

In predicate calculus formulae are built from atoms consisting of a predicate and arguments. The number of arguments (and in case of typing also their types) are fixed for each predicate. Each predicate, in interpreted, determines a relation consisting of those tuples of domain values which satisfy the predicate.

Horn clauses are logical formulae equivalent to an implication of the form $At_1 \& \dots \& At_n \Rightarrow At$ where At and the At_i are atoms. These clauses have the following "constructive" interpretation: if the predicates At_1, \dots, At_n are all satisfied then so is the predicate At . Thinking in terms of relations such a clause will mean that given that some rows occur in some relations another row must occur in another relation. The clause becomes "tuple-generating" information.

The above constructive interpretation has paved the way to the *production rule* approach in artificial intelligence and has found its concrete implementation in the programming language Prolog [CM81]. In this language the above implication is written in reverse order $At \Leftarrow At_1 \& \dots \& At_n$.

The atom A_t is called the *head* and the conjunction $A_{t1} \& \dots \& A_{tn}$ is called the *body* of the clause. The arguments can be constants, variables or even arbitrary terms and lists. The proper understanding of the variable occurrences is obtained by considering the variables occurring in the head to be universally quantified, whereas variables which occur only in the body should be quantified existentially. Under this interpretation a single clause denotes a large set of implications obtained by substitution of constants for variables. The number n of atoms in the body can be zero; in this case the head of the clause is unconditionally true. Such clauses are called *facts*.

The essence of logic programming is that a program will consist of a set of Horn clauses. On the one hand this program can be seen as a declarative statement of a number of implications. On the other hand the program can be used for producing a database consisting of facts if we start with the facts in the program and use the other clauses to derive new facts from existing ones, continuing this process until no new facts can be derived. In practice this process is initiated by asking a specific *question* : starting with a given predicate and a given list of arguments (the *goal*) one looks in the program for a clause of an applicable rule (*matching under unification*) and replaces the goal by the finite (possibly empty) list of *subgoals* obtained from the body of the clause used by performing the substitutions required by the unification. This process is repeated till no subgoals remain and the goal is satisfied, or till all possibilities have been exhausted and the goal fails.

In Prolog the above goal oriented evaluation strategy has been implemented. The evaluation process has been made deterministic by requiring a top to bottom and left to right order in the use of rules and the satisfaction of conjuncts in combination with backtracking.

It has been observed by several researchers that there is a close relation between the natural join operation from relational databases and the evaluation of bodies of Horn clauses. Compare the relation described by the Horn clause

$$Q(X,Y) \Leftarrow P(X,Z) \& R(Z,Y) ,$$

and the view definition:

$$Q(X,Z) = \text{proj}(P(X,Z) \text{ join } R(Z,Y))[X,Y] .$$

This analogy has led to the so called *Compiled Approach* proposed by Chang, Reiter and Gallaire & Minker [C78, R78,GMN84]. The idea here is to "compile" Logic programming questions into database queries, according to the translation rules: Facts -> rows in base tables; Rules -> view definitions, where conjunction becomes a join and multiple rules for the same head becomes a union, and finally Questions -> queries. That this idea indeed will lead to a compilation of a subset of Prolog into a Database Language, provided the database system is

