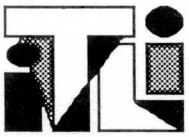


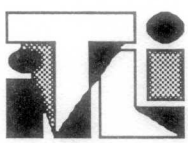
Instituut voor Taal, Logica en Informatie

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. Idealy, 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** <attributenamelist> **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 At1 & ... & Atn => At where At and the Ati are atoms. These clauses have the following "constructive" interpretation: if the predicates At1, ... , Atn 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 is 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 <= At1 & ... & Atn .

The atom At is called the *head* and the conjunction At1 & ... & Atn 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) <= P(X,Z) & R(Z,Y) ,

and the view definition:

Q(X,Z) = **proj**(P(X,Z) **join** R(Z,Y))[X,Y] .

This analogy has lead 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

sufficiently powerful and its language has the required expressive tools has been established in [VV86].

One of the problems faced by the compiled approach is recursion. Recursive clauses will lead to recursive view definitions, and these are in general not allowed in a database. On the other hand there exists a semantically plausible solution for this problem. One needs to compute the solution of a fixed point equation in the relational algebra. Provided the recursive system is monotone in all arguments existence of a minimal fixed point is guaranteed by the Krasner Kuratowski theorem. If the system is moreover continuous the fixed point is obtainable as the limit of a countable chain of constructive approximations obtained by k-fold expansion of the recursive view definitions, replacing the then remaining recursively defined relations by corresponding empty relations which play the role of the bottom element in the domain of relations ordered by inclusion. This approach to the semantics of recursion is well known in the area of semantics of programming languages [M74,dB80]. Its applicability in the context of the compiled approach is also known. The problem is rather to obtain a workable practical implementation of this idea; see for example [NH84,U85,VV86].

In order for such a translation to be manageable the collection of Horn Clauses which can be compiled is severely restricted. Arguments of predicates should be either variables or constants, so no terms can be used, and this reduces the unification part of the Prolog evaluation mechanism to an almost triviality. This restriction excludes also the use of lists as arguments. Also many elements of non-pure Prolog, like build-in predicates, asserts and retracts and the cut-operator are excluded.

In **RL** we want to combine the experienced advantages of logic programming rules with a clear relational semantical framework. Therefore we include a fragment of horn clauses comparable to the Prolog fragment dealt with in the compiled approach as defined in [VV86]. We do include, however, recursive rules.

The relations determined by clauses will be called *clausal relations*. A single clause will provide part of the definition of some clausal relation. This is not a complete meaning for this relation since there may be other clauses with the same predicate in the clause head. Given the possibility of recursion, the total collection of clauses in a program has to be considered at once in order to interpret is as a recursive system, the solution of which assigns meanings to all clausal relations involved.

The clausal relation has a name equal to the name of the corresponding predicate in the head. attribute names and types are determined by the variables used in the clausal rules. As a consequence in all occurrences of a predicate the same identifiers should be used at the same positions, except for explicit renamings. The types of the attributes are either inferred from earlier occurrences of the corresponding attribute names, or explicitly determined by some type declaration.

3.2.1 Syntax of Clauses.

We next turn to the syntax of clauses:

<clause> ::= <clause_head> **when** <clause_body> | <clause_head>
<clause_head> ::= <relationname> (<arglist>)
<clause_body> ::= <conjunct> | <conjunct> **and** <clause_body>
<conjunct> ::= <clause_head_invocation> | <arithmetic_relation>
<clause_head_invocation> ::= <clause_head>
<arg> ::= <attributename> | <substitution>
<substitution> ::= <variablename> / <attributename> |
<constant_expression> / <attributename>

An example of a clause:

exempt (part , order) **when** subpart (part / fragment , whole) **and**
exempt (whole / part , order)

Clauses can be analyzed as implications of the form Y1 & Y2 & .... & Yk => X . Here the X is called the clause-head and the Yi are called conjuncts. There is no negation permitted. Syntactically a clause-head is a relationname with a list of arguments; these arguments either are attributes or substitutions; in the later case either another attribute-name or a constant expression is substituted for the attribute-name belonging to the relation-name in the head. Conjuncts in clauses may invoke other clausal relations, including the one defined in the head (leading to recursion); it is also permitted to invoke a tabular relation (our syntax does not specifically specify this possibility, but it does not prohibit it either), or even a restricted form of a constraint (an unconditional arithmetical relation). The idea to allow this interaction between clauses and arithmetical constraints was inspired by EQLOG [GM84].

3.2.2 Semantics of clauses.

The semantics of clausal relations is determined by solving the least fixed point of a system of inclusions in the relational algebra. Unknowns in this system are the clausal relations. There exists a clausal relation corresponding to each relation name appearing in a head of a clausal rule. For each name of a clausal relation there is a fixed number of attributes with fixed names which determines the Cartesian product of domains which contains this relation.

A clause-head(-invocation) describes a selection from the corresponding clausal relation of those rows where:

-- if some constant expression is substituted for an attribute then the denotation of that constant expression is the value of that attribute,

-- if for a pair of attributes the same variable is substituted then the corresponding attributes have equal values.

An arithmetic relation describes the relation consisting of those tuples of elements in the domains corresponding to the attributes occurring in the relation for which the relation evaluates to **true**.

A clause body describes the natural join of the relations described by its conjuncts. The occurrence of common columns on which the join is based is determined by the attribute names and the variable names substituted for them by explicit renamings.

A clausal rule determines the inclusion that the relation described by its body is included in the relation described by its head. In order to make this inclusion feasible the relation described by the body must conform the relation in the head. Attributes which don't occur in the head of the corresponding rule are projected away; attributes which occur in the clause head but not in the body are introduced in the body-relation by forming the Cartesian product with a filling relation over the missing attributes; this filling relation consists of all rows having a constant value for an attribute if such a constant is specified in the head, and an arbitrary value in the corresponding domain otherwise.

Since all relational operations used in the description of the above system of inclusions are monotone and continuous the standard techniques show that the least fixed point solution can be evaluated as the limit of a countable chain of approximations obtained by starting with empty clausal relations, evaluation of the bodies and performing the required inclusions and repeating this process.

3.3 Constraints.

The third sort of rules we allow in **RL** are the constraints. Here the typical examples are algebraic equalities and inequalities. An equation $x + y = z$ represents a relation consisting of those tuples of numbers x,y,z for which the equation holds. An inequality in the same way denotes a relation. Taking such an Algebraic Geometric perspective such (in)equalities can be seen as hypersurfaces and half spaces (the set of all points in space where a given polynomial is nonnegative) within a Cartesian Product of several copies of the real numbers.

If there is more than one constraint the intended meaning is that all constraints should be enforced. Hence, the meaning of the system of constrains becomes the intersection of the corresponding hypersurfaces and half spaces. Since different constraints will contain different variables these algebraic sets first should be included together in some large Cartesian product involving a coordinate space for each variable occurring in some constraint. For example the

relation described by the two constraints x + y = z and t = y * z consists of all tuples x,y,z,t satisfying both equations, so this represents the intersection of two hypersurfaces in four dimensional space. This is exactly the way intersections are performed in the relational framework, provided we take the variables for attribute names. Once having established this analogy we can extend it by allowing a far larger collection of constraints. In **RL** we will allow conditional equations and inequalities, and we will also allow to invoke any tabular or clausal relation whose meaning has been established before as a constraint.

Note the following difference between the tabular and clausal relations encountered before and the relation defined by the intersection of all constraints here: Both tabular and clausal relations have a name assigned to them at the time of definition. They are *Named Relations*. The relation determined by the constrains has no name assigned to it by its definition; it is supposed to represent the *world in its relevant aspects*. We will call this relation the *Principal relation*. In a context where there is a single collection of rules this Principal relation needs no name. However, in a modularized version of **RL** this lack of a name may become a source of confusion and then the Principal relation will obtain the name of the *Rules Module* which defines it. This also creates the possibility to invoke the principal relation of one module as a clause-head-invocation within another module which depends on the first one.

The attributes of the Principal relation are all attributes which occur in any constraint in the module. Their types are either inferred from the program or specified explicitly by some type declaration.

3.3.1 Syntax of constraints.

Constrains are described by the following syntax:

<constraint> ::= **if** <condition> **then** <arithmetic_relation> <elifsequenceoption>
else <arithmetic_relation> **fi** |
<arithmetic_relation> | <condition>
<elif> ::= **elif** <condition> **then** <arithmetic_relation>
<condition> ::= <clause_body>
<arithmetic_relation> ::= <arithmetic_expression> <relop> <arithmetic_expression>

Examples of constraints:

totalprice = unitprice * quantity *(1 + taxrate)
if exempt(part,order) **then** taxrate = 0 **else** taxrate = percentage **fi**
category(part / object,class) **and** vat_rates(class,percentage)

Constraints are built from arithmetic relations and conditions and resemble Boolean expressions. The syntax suggests that the arithmetic relations are equations and inequalities of numeric expressions, but this is not necessarily the case, since the mechanism of abstract data types allows the programmer to introduce new relops (relational operators) on new types (EG., comparing dates in order to find out if one date precedes another in time).

The syntax allows for long conditional constraints and simple unconditional ones; it also allows for a naked condition to be enforced as a constraint. We included this possibility because it is a more natural way for expressing something which would be expressible anyhow: the condition C is equivalent to the degenerate conditional constraint:

if C **then** 0=0 **else** 0=1 **fi** .

Attributes which occur in some constraint will become attributes of the principal relation. If we invoke named relations like tabular relations or clausal relations in conditions, these named relations will have their own attributes which in general are different from those of the principal relation. This can be achieved by renaming of attributes in the conditions. It is not required that auxiliary relations and the principal relation have disjoint sets of attributes; they may share a common attribute but in that case the type of the shared attribute must be equal for both types of relations.

3.3.2 Semantics of constraints.

The system of constraints in a program describes a single relation over the set of attributes formed by all attributes which occur in any constraint. This relation consists of all tuples of values which, if substituted for the attributes make all constraints if evaluated as a Boolean expression, evaluate to **true**.

4 Rules modules and their meaning.

After the above treatment we now are able to explain what a (single module) **RL** program will be and how it determines a meaning. This program will consist of a single *rules module* which consists of a heading (providing type declarations if needed) and a sequence of rules. Rules are tabular rules, clauses or constraints.

4.1 Overall Syntax.

The global structure of a program is described by:

<program> ::= <package_sequence>
<package> ::= <Abstract_datatype_module> | <rules_module>

but for the time being we will consider only the case where the program contains a single rules module, possibly preceded by some Abstract datatype modules. Both the syntax and the semantics of Abstract Datatype modules are left unconsidered in this paper. The syntax of a rules module is:

<rules module> ::= <rmhead><rules_block_option> **erules**
<rmhead> ::= **rules** <using_slot_option><about_slot_option> **in**
<using_slot> ::= **using** <abstract_datatype_name_list>
<about_slot> ::= **about** <attribute_def_list>
<attribute_def> ::= <typename> : <attribute_name_list>

<rules_block> ::= <rule_list>
<rule> ::= <tabular_rule> | <clause> | <constraint>

The above rules describe the global structure of a rules module as a heading, followed by a sequence of rules of the three types introduced in the previous section. The purpose of the using-slot in the heading is to import the abstract data type modules which are used in order to specify the domains and operations used in the rules in the module. In a modularized version one can also import other rules modules. The about-slot gives a type specification of those attributes whose type should be unknown otherwise. In the context of a program consisting of a single module this amounts to giving a type specification for all attributes. In the modularized version attributes preserve their type under import, and it will no longer be necessary to specify the types for all attributes.

From the above description it can be inferred that the order in which the rules of a single module are given is almost irrelevant for its meaning. The only requirement on the textual order of a rule is that all relations used in the definition of a tabular rule have been defined in an earlier tabular rule. On the other hand it may be a convenient strategy to write down the rules in the order: tabulars, clauses and constraints.

4.2 Semantics of a Rules Module.

The meaning of a Rules module is obtained in three steps.First interpret the tabular rules ; this is possible since they are single line definitions in terms of relations which are either explicitly known or have been defined before.

Based on the tabular relations we interpret the clauses as a (possibly recursive) system. We solve this system and thereby assign meanings to all clausal relations. Next we use both the clausal and the tabular relations in order to interpret all constraints, and thus determine the Principal relation as the intersection of the corresponding hypersurfaces and half spaces.

The meaning of the entire module consists of the collection of all named auxiliary relations (the tabular and the clausal relations) and the principal relation.

5. Modularization.

The purpose of modularization is the decomposition of large systems into manageable parts. These parts may depend on each other, but within some part you shouldn't have to be aware of details of some imported other part, unless it is explicitly required that you have access to these details. Modularization and *encapsulation* come together. However, in the **RL** language as proposed in [V85] I have not introduced an encapsulation mechanism since it is not clear what you would want to hide inside a module by making it invisible from outside. So in principle, if you import module A in module B , all tabular rules, clauses and constraints in A become visible and known inside B . This has the following consequences:

-- names used in A for relations and attributes cannot be used freely in B , unless they are replaced by a renaming during import. Such renamings are provided, but they in turn lead to syntactic and semantic problems, in particular if the import relation is not structured like a tree but like a general directed acyclic graph.

-- tabular relations defined in A preserve their meaning in B, since they cannot be refined. Clauses in A may have to be reinterpreted since B may contain new rules for clausal relations defined in A . For example we may use clauses for defining holidays, and then module A can represent holidays common in the world, whereas B talks also about the Dutch holidays. Since all clauses may depend on each other the entire system of clauses must be reevaluated.

-- constraints in A remain constraints in B . So in particular their attributes become attributes of the principal relation of B .

I am inclined to accept the first two consequences but not the third. This would lead to a situation where modules which have imported many ancestors would have very large sets of attributes for their principal relation. The principal relation of B is intended to represent the world in its relevant aspects according to B . So the module B itself should state what it considers to be relevant. As long as B contains no reference to some tabular or clausal relation in A this relation won't affect the semantics of B. But its principal relation would always be affected by the import.

Therefore I will take as attributes for the principal relation of B those attributes which are explicitly listed in constraints inside B . If the programmer wants to include all attributes of the

principal relation of A he can obtain this effect by enforcing A's principal relation as a constraint by invoking A as a clause-head-invocation; this is possible since A's principal relation is a named relation in B . If the programmer does not want to include all attributes of A's principal relation this does not mean that the information of A is entirely disregarded. Those attributes which A's and B's principal relation are sharing will be restricted in their possible values according to what the principal relation of A allows. So the semantic effect of importing A on the principal relation of B is that the principal relation of A is projected on the common attributes, and this projection is enforced as a constraint in B .

6. Conclusion.

We have presented the outlines of a system which integrates three rather distinct modern technologies for intelligent systems which share the property of having a relational model, but which otherwise have very little in common. Due to lack of space I cannot include a convincing example which will show that the **RL** language will be a convenient tool for expressing real life rules, but my experience is positive. Also the modularization feature turned out to be useful in the examples tried.

The real question is whether an **RL** system as described in this paper can be implemented. Given the fact that partial combinations of the elements of **RL** have been implemented I am positive on this issue. An integration of tabular rules and clauses is exactly the theme of the compiled approach discussed in section 3.2 which we know to be feasible on a sufficiently powerful database [VV86]. We also know how to deal with arithmetic equations and inequalities in isolation. Hence some combination of an equation solver and a powerful database seems to be an answer. The example of EQLOG [GM85] shows that a combination of clauses and abstract datatypes is feasible. Aggregates have not been implemented at the proposed level of generality but most database systems provide for the more common types by means of ad hoc features.

Acknowledgements.

This paper contains the essential ideas behind the design I made during my visit at the IBM San Jose Research Center in the period Jan-Aug 1985. I am indebted both for the subject and for many of the specific design issues to Peter Lucas with whom I have had many discussions on larger and smaller features in **RL**. Some of those features which troubled him are not included in the present paper, but this is an indication of the fact that they are hard to explain convincingly in 15 pages; it does not mean that I have given up on them.

The relational approach for integration of databases and logical rules is clearly inspired by the project performed in cooperation between IBM INS/DC in Uithoorn and the University of

Amsterdam during the past 30 months. I am grateful for the continuing support of A.J. du Croix in promoting this project and to C.F.J. Doedens and S.J.C. Elbers for building the implementations and to Ghica van Emde Boas for originating this project.

Theo Janssen and Leen Torenvliet provided crucial criticism which forced me to uncover the bare essentials of the **RL** proposal in a structured way.

References.

C78 Chang, C.L., *DEDUCE 2: Further investigations of Deduction in Relational Databases,* **in** [GM78].

CM81 Clocksin, E.F. & C.S. Mellish, *Programming in Prolog*, Springer 1981.

dB80 de Bakker, J.W., *Mathematical Theory of Program correctness,* Prentice Hall, 1980.

dB84 de Brock, E.O., *Database Models and Retrieval Languages*, PhD Thesis Technical University Eindhoven, Mrt 1984.

GM78 Gallaire, H.V. & J. Minker, eds., *Logic and Databases*, Plenum, New York 1978.

GMN85 Gallaire, H.V., J. Minker & J-M. Nicolas, *Logic and Databases: a Deductive Approach*, Computing Surveys **16** (1985) 153-185.

GM85 Goguen, J.A. & J. Messeguer, *Equality, Types, Modules, and (why not) Generics for Logic Programming*, J. Logic Programming **1** (1985) 179-210.

IL84 Imielinski, T. & W. Lipski, jr., *The Relational Model of Data and Cylindric Algebras*, J. Comput. Syst. Sci. **28** (1984) 80-102.

L80 Lucas, P., *On the Structure of Application Programs*, **in** D. Bjørner, ed., *Abstract Software Specifications*, Proc. Copenhagen Winterschool 1979, Springer LCS **86** (1980) 390-438.

Lu85 Lucas, P., *On the Versatility of Knowledge Representations*, **in** E.J. Neuholt & G. Chroust, eds., *The Role of Abstract models in Information Processing*, Proc IFIP Working Conference, Vienna Jan 1985, North Holland Publ. Cie. 1985.

M74 Manna, Z., Mathematical Theory of Computation, Computer Science Series, McGraw-Hill, New York 1974.

NH84 Naqvi, S.A. & L.J. Henschen, *On Compiling Queries in Recursive First-Order Databases*, J. ACM **31** (1984) 47-85.

R78 Reiter, R., *Deductive Question-Answering on Relational Databases*, **in** [GM78].

U82 Ullman, J.D., *Principles of Database Systems*, 2nd ed. Computer Science Press, Rockville MD 1982.

U85 Ullman, J.D., *Implementation of Logical Query Languages for Databases*, ACM Trans. Database Systems, **10** (1985) 289-321.

V85 van Emde Boas, P., ***RL**, a Language for Enhanced Rule Bases Database Processing*, Working Document, Rep RJ 4869 (51299), Oct 1986.

VV86 van Emde Boas, H. & P. van Emde Boas, *Storing and Evaluating Horn-Clause Rules in a Relational Database*, IBM J. Res. Develop. **30** (1986) 80-92.

vW75 van Wijngaarden, A. ea., *Revised Report on the Algorithmic Language ALGOL68*, Acta Informatica **5** (1975) 1-236.