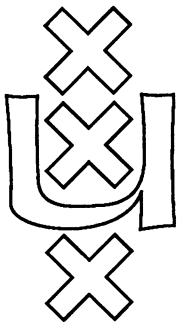
Institute for Language, Logic and Information

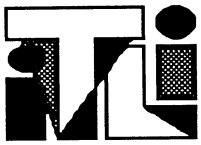
THE RULE LANGUAGE RL/1

Sieger van Denneheuvel

ITLI Prepublication Series
for Computation and Complexity Theory CT-89-10



University of Amsterdam



Instituut voor Taal, Logica en Informatie
Institute for Language, Logic and Information

Faculteit der Wiskunde en Informatica
(Department of Mathematics and Computer Science)
Plantage Muidergracht 24
1018TV Amsterdam

Faculteit der Wijsbegeerte
(Department of Philosophy)
Nieuwe Doelenstraat 15
1012CP Amsterdam

THE RULE LANGUAGE RL/1

Sieger van Denneheuvel
Department of Mathematics and Computer Science
University of Amsterdam

Received December 1989

The rule language RL/1

Sieger van Denneheuvel
Department of Mathematics and Computer Science
University of Amsterdam

November 30, 1989

Abstract

In this report we introduce the rule language **RL/1** as an intermediate step towards implementing the declarative rule language **RL**. The language **RL/1** is an effort to integrate logic and functional programming with relational databases and constraint solving into a relational framework.

1 Introduction

The language **RL/1** described in this report is intended to be an intermediate step towards implementing the declarative rule language **RL** (see [VEMD86a], [VEMD86b] and [VEMD86c]). The report [ROES87] has influenced the definition of **RL/1** strongly. Some of the problems noted there, in particular the notation for supplying arguments to rule invocations, are considered. The language represents an effort to integrate logic and functional programming with constraint solving and relational databases. Towards the user, the ease of logic programming constitutes a declarative and user friendly knowledge base. On the other hand query processing with help of a relational database and a constraint solver, ensures that large amounts of data can be processed effectively. In the current version of **RL/1** we do not focus on recursion as is done in the NAIL! system [ULL89] and the **LDL** logical database [NAQV89] but rather on the integration of a constraint solving subsystem with a relational database system (see [DEN88a] and [DEN88b]).

Associated with the rule language **RL/1** is a database language **DL**. The database language acts as a parameter for **RL/1** (c.f. [DEN89]) and models the particular database system that is being used for evaluation of queries. In this report the intended relational database is **SQL**. A compiler for the language **RL/1** has been implemented in a prototype system. In addition a constraint solver and a relational database were coded directly in PROLOG.

Outline of the rest of this report. In section 2 we give an overview of the logical objects that can be defined in **RL/1**. The remaining sections describe the language hierarchically starting with the lower syntactic categories. In section 3 we give the database language **DL**. Sections 4 to 9 describe the rule language **RL/1**.

Acknowledgements. I thank Peter van Emde Boas, Karen Kwast and Gerard Renardel for useful criticism and remarks. Otto Moerbeek is gratefully acknowledged for teaching me the dirty tricks of LaTeX.

2 Logical objects in RL/1

In **RL/1** logical objects can be created, destroyed, evaluated and otherwise manipulated. Logical objects come in four types: tables, maps, clauses and functors. A property that logical objects share is that they have attributes and denote a relation. For a particular object the denoted relation can be requested with use of a LIST command (see the section on query commands below). From the logical objects tables and maps are *extensional objects* in the sense that the relation denoted by the object is stored explicitly in the relational database. Clauses and functors on the other hand are *intensional objects* whose relations can be materialized by evaluation of the definition of the object.

A second distinction for logical objects is that they are either relational or functional. Tables and clauses are *relational objects* and are comparable to tables and views of a relational database. In **RL/1** relational objects are used in rule expressions. Maps and functors on the other hand are *functional objects* and are invoked in scalar expressions. Functional objects in **RL/1** are more general than mathematical functions since for one assignment of argument values a functional object may return several values. In this respect functional objects behave in a similar way as relational objects. The difference between functional and relational objects is only that the first kind is used in functional notation and the latter is invoked as a predicate. The above properties are summarized in figure 1.

3 The database language DL

The database language of **RL/1** is dependent on the underlying relational database. Since queries are eventually executed on the relational database, operators from the database language **DL** need to be supported by the relational database language. Also the language **DL** should be rich enough so that features which are available in the database can be accessed from the **RL/1** system directly. The syntax for the database language **DL** and the rule language **RL/1** was chosen so that it can be parsed efficiently by the operator oriented parser of PROLOG (see [CLOCK81]). On the other hand it is also parsed efficiently with a conventional programming language (such as 'Pascal'

logical-obj	relational-obj (rule-exp)	functional-obj (scalar-exp)
extensional-obj (EDB)	table	map
intensional-obj (IDB)	clause	functor

Figure 1: Classification of logical objects.

or 'C'), since a look-ahead of one token is sufficient for parsing the syntax. For our notation we follow [DATE87]:

- Special characters are written as shown and keywords are denoted in uppercase. Material in lower case represents a syntactic category.
- Vertical bars "|" are used to separate alternatives.
- Square brackets "[" and "]" are used to indicate that the material enclosed is optional. It consists of a set of one or more items (separated by vertical bars) from which at most one is to be chosen.
- Braces "{" and "}" are used to indicate that the material enclosed consists of a set of several items (separated by vertical bars) from which exactly one is to be chosen.
- In case one of the meta characters "[","]","{","}" and "|" is used as a normal character, it is enclosed by double quotes.
- If "x" is a syntactic category then "x-list" is a category consisting of a comma separated list of one or more "x"s.
- If "x" is a syntactic category then "x-colonlist" is a category consisting of a colon separated list of one or more "x"s.
- A percent sign "%" introduces a comment in text generated by the grammar. The scope of the comment is from the percent sign upto the end of the line.

The language **DL** has two scalar types, namely NUMBER and STRING. The syntactic category 'numeric-val' represents constants of type NUMBER and the category

```
numeric-fac
    ::= [ - ]  numeric-val | functor-invoc | map-invoc
    |   [ - ]  numeric-func | variable | ( numeric-exp )
numeric-term
    ::= numeric-fac
    |   numeric-term  * | /  numeric-fac
numeric-exp
    ::= numeric-term
    |   numeric-exp  + | -  numeric-term
string-term
    ::= string-val | functor-invoc | map-invoc
    |   string-func | variable
string-exp
    ::= string-term
    |   string-exp CAT string-term
scalar-exp
    ::= numeric-exp | string-exp
```

Figure 2: Grammar of scalar expressions.

'string-val' denotes constants of type STRING. The syntax of scalar expressions is defined in figure 2.

In the grammar 'functor-invocation' and 'map-invocation' are calls to functional objects. Both functor and map objects are described in the sections below. The current implementation restricts the use of functors and maps in constraints. Functor and map invocations are allowed in a constraint only if the constraint does not include any of the $\vee$ or $\neg$ operators. The above grammar is incomplete with respect to the categories 'string-func' and 'numeric-func', since these functions depend on the actual **SQL** database used for query evaluation. The syntax of constraints is defined in figure 3 (standard predicates are not listed).

Null values are supported by **SQL** and therefore also by **DL**. A scalar expression is considered to evaluate to NULL if any of the variables occurring inside the expression evaluate to NULL. The constant NULL (of type NUMBER or STRING) was not added to scalar expressions in **DL** because the use of NULL inside scalar expressions is not supported by **SQL** (see [DATE87]). On the other hand the presence of NULL values in the extension of objects is supported by **SQL** and therefore also by **DL** (see the section on object extensions). The connectives of **DL** are based on a three valued logic ([DATE89]) and are defined by the truth tables given in figure 4. The constants TRUE and FALSE can be defined in **DL** with use of clauses (see the section on clausal objects below). The constant UNKN denotes the null value in the domain of truth values and can not be defined in **DL** since NULL values are not allowed in scalar expressions.

```
constraint-fac
    ::= [ ¬ ] { standard-pred | ( constraint ) }
    |   [ ¬ ] numeric-exp { = | < | > | ≤ | ≥ | ≠ } numeric-exp
    |   [ ¬ ] string-exp { = | < | > | ≤ | ≥ | ≠ } string-exp
constraint-term
    ::= constraint-fac
    |   constraint-term ∧ constraint-fac
constraint
    ::= constraint-term
    |   constraint ∨ constraint-term
```

Figure 3: Grammar of constraints.

¬	
FALSE	TRUE
TRUE	FALSE
UNKN	UNKN

∧	TRUE	FALSE	UNKN
TRUE	TRUE	FALSE	UNKN
FALSE	FALSE	FALSE	FALSE
UNKN	UNKN	FALSE	UNKN

∨	TRUE	FALSE	UNKN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKN
UNKN	TRUE	UNKN	UNKN

Figure 4: Truth tables of **DL** propositional connectives.

```
value
    ::= numeric-val | string-val | NULL
row
    ::= "[" value-list "]"
column
    ::= "[" value-list "]" | value | EXTERN
relation
    ::= "[" row-list "]" | row | EXTERN
```

Figure 5: Grammar of object extensions.

4 The rule language RL/1

In contrast with the database language the rule language is not dependent on the underlying relational database. The operators of the rule language therefore do not correspond directly with operators of the underlying relational database language as was the case for the database language **DL**. Instead the rule language has a declarative nature and specifies what is known to be true in a domain of interest and does not specify how the knowledge is to be used for evaluation of a particular query. As we mentioned before there are four types of logical objects:

```
logical-obj
    ::= table | map | clause | functor
```

4.1 Object extensions

An extensional object can be initialized by an explicit listing of its values when it is created. A tuple is represented by the syntactic category row and consists of a list of values. The category relation stands for a set of tuples and in case the relation is a singleton set a pair of square brackets may be omitted. A column is a set of tuples consisting of one value only. Moreover all values in a column are of the same type (either NUMBER or STRING). In case the column has only one value a pair of square brackets may be omitted. Both relations and columns can be external indicating that the extension of the object is already stored in the database. The syntax of extensions is given in figure 5.

4.2 Rule expressions

Rule expressions support the definition of intensional objects and are constructed from table and clause invocations and constraints. Table and clause invocations are described below in the sections on tabular and clausal objects respectively. The syntax of rule expressions is defined in figure 6. The declarative AND and OR operators used

```
rule-fac
    ::= [ NOT ] { clause-invoc | table-invoc }
    |   [ NOT ] { constraint | ( rule-exp ) }
rule-term
    ::= rule-fac
    |   rule-term AND rule-fac
rule-exp
    ::= rule-term
    |   rule-exp OR rule-term
```

Figure 6: Grammar of rule expressions.

in rule expressions are more general than the corresponding operators join and union in relational algebra (see also [HANS89]). Operands of the AND (and OR) operator are not restricted to be finite relations as in relational algebra, but may be constraints or clause invocations that, by themselves, can represent infinite relations. The question whether a rule expression as a whole represents a finite or infinite relation is answered with help of a constraint solver (see [DEN88b]). Also operands of the OR operator are not restricted to have the same attributes as is usual for the union operator in relational algebra.

4.3 Positional notation and attribute notation

Both *positional notation* and *attribute notation* can be used to supply arguments for invocations of extensional and intensional objects (cf. the language **LDL** in [NAQV89] where attribute notation can not be applied on intensional objects). Each notation has its own advantages and disadvantages. We think that the user will prefer positional notation for objects with few attributes and for objects with many attributes, attribute notation is preferred. A mixture of positional and attribute notation is not allowed in **RL/1** since this probably would reduce the readability of the program code.

In positional notation the number of arguments supplied in an object invocation is equal to the number of attributes in the definition of the object. The positions of the arguments match the positions of the attributes in the object definition. As a consequence the attributes of the object definition need not be specified in the object invocation. A disadvantage is the need to specify all arguments. Contrary to positional notation in attribute notation the number of arguments supplied in an object invocation may be less than the number of attributes in the definition of the object. With each argument in the object invocation also the name of the substituted attribute is given. A clear advantage of attribute notation is that only the relevant attributes appear in the invocation. The disadvantage is that each argument carries an extra attribute specifier. The syntax for arguments is defined in figure 7.

In positional notation, some abbreviations are allowed. The anonymous variable '#'

```
assignment
    ::= attribute = scalar-exp
positional-arg
    ::= scalar-exp | #
attribute-arg
    ::= assignment | attribute
arguments
    ::= positional-arg-list
    |   "[" attribute-arg-list "]" | *
```

Figure 7: Grammar for arguments.

	Positional notation	Attribute notation	Relation
(a)	t(*)	t([a,b,c])	a b c 1 2 3
(b)	t(x,y,#)	t([a=x,b=y])	x y 1 2
(c)	t(x+1,y,#)	t([a=x+1,b=y])	x y 0 2
(d)	t(3,y,#)	t([a=3,b=y])	y void

Figure 8: Positional and attribute notation.

can be used as a don't care variable in object invocations and is similar to the variable '_' in PROLOG. Anonymous variables can stand for numeric or string variables and are not listed in answer relations. Attribute notation is denoted by surrounding the invocation arguments with an extra pair of '['...']' brackets. Each attribute argument contains the substituted attribute followed by '=' and a scalar expression in case of an assignment. When both sides of '=' are equal, assignments can be abbreviated to a single attribute. Attribute notation is in a way equivalent to positional notation and therefore substitutions of general scalar expressions for object attributes are allowed. As a special case (in analogy to **SQL**) also a star notation '(*)' is available to denote all attributes of an object. In figure 8 positional queries are shown next to equivalent queries in attribute notation ('t' is a relation with attributes <a,b,c> and one tuple [1,2,3]). In query (c), 1 was subtracted from the attribute value for attribute 'a' to yield the correct answer for 'x'. In (d) the substituted value 3 for attribute 'a' functioned as an implicit selection and the answer relation is empty.

```
object
    ::= logical-obj | module
domain-decl
    ::= NUMBER : variable-colonlist
    |   STRING : variable-colonlist
    |   DOM logical-obj :  variable-colonlist
property-decl
    ::= KEY : variable-colonlist
    |   NOTNULL : variable-colonlist
    |   PRIVATE : logical-obj-colonlist
using-decl
    ::= USING : module-colonlist
declaration
    ::= domain-decl | property-decl | using-decl
module-decl
    ::= MODULE module [ ( declaration-list ) ]
module-term
    ::= CLOSE
```

Figure 9: Grammar of module declarations.

4.4 Module definitions

A module starts with a module declaration and is ended with a module terminator. In **RL/1** modules are also considered to be objects. A property that module objects share with logical objects is that they have attributes. The attributes of a module object are the variables declared in the module. A module declaration consists of a module name followed by a list of declarations. The declaration list is the only place in the module where variables can be declared (see figure 9).

Each variable appearing in the module is declared with a NUMBER, STRING or DOM domain declaration. In the first two cases the domain of the variable is as indicated. In the last case the domain is defined by the logical object (see the section on domain objects below). In addition to a domain declaration, a variable can have one or more property declarations. The property declarations listed in the grammar above for variables are useful for the definition of tables and maps. The USING declaration allows logical objects defined in other modules to be imported, so that they become available in the declared module. If the imported module also refers to other modules via a using declaration, these modules are also imported in the declared module. For encapsulation the PRIVATE declaration applied on a logical object ensures that the object is only visible inside the module. Other modules are not allowed to use or modify private objects.

```
MODULE enumexamples(STRING:x).
ENUM sizes(x) = medium.
ENUM colors(x) = [red,green,blue].
ENUM animals(x) = EXTERN.
SHOW colors(x).  % yields 'red','green','blue'
CLOSE.
```

Figure 10: Enumeration objects.

5 Tabular objects

Tabular objects correspond to base tables of the underlying relational database. The TABLE command creates a table with attributes in the order of the given attribute list. The new table is initialized with a relation and in case the relation is omitted the table is left empty. A table can be declared as an external relation by using the EXTERN option so that the table may already be filled with a large number of tuples:

```
table-def
    ::= TABLE table ( attribute-list ) [ = relation ]
table-invoc
    ::= table ( arguments )
```

A tabular (or map) object is defined with a *single* definition. If a new tabular object is defined subsequently with the same name, the original object is redefined to the new object. For modules the situation is different. If a tabular object is defined in both modules A and B and module B imports module A then the object in A becomes invisible.

5.1 Enumerations

The ENUM command creates an enumeration as a tabular object with one attribute. The new enumeration is initialized with the column if present. Enumeration definitions have the following syntax:

```
enum-def
    ::= ENUM table ( attribute ) [ = column ]
```

Some examples of enumerations are listed in figure 10. The SHOW query can be used to evaluate a rule expression and is described in the section on query commands.

```
MODULE mapexamples(NUMBER:x,STRING:y).
MAP map1(x,y) = [[1,red],[2,green],[2,yellow],[3,blue]].
MAP map2(x,y) = [1,red].
MAP map3(x,y) = EXTERN.
PRINT map1(1).  % yields 'red'
PRINT map1(2).  % yields 'green' and 'yellow'
CLOSE.
```

Figure 11: Map objects.

6 Map objects

The definition of maps is similar to the definition of tables and the remarks made in the section on tabular objects also apply for maps. However the invocation of a map is different from a table since a map is used in functional notation in scalar expressions (as mentioned before). The syntax for maps is as follows:

```
map-def
    ::= MAP map ( attribute-list ) [ = relation ]
map-invoc
    ::= map [ ( arguments ) ]
```

The MAP command creates a map object with attributes in the order of the attribute list. The map is initialized with the relation if present. Map objects can be declared as external relations in the same way as tabular objects. The last attribute of the attribute list is the return variable which yields the function value of the map. A map that is defined with n attributes is called in a map invocation with n-1 arguments. Figure 11 lists some examples of maps. The PRINT query is used to evaluate a scalar expression and will be described in the section on query commands.

6.1 Vectors

The declaration of variables in a module declaration does not allocate storage. In this section variables will be described, called vectors, that do allocate storage. Vectors are persistent, in the sense that their value and name are stored in the relational database. Also vectors can be assigned more than one value. The syntax for the creation of vectors is as follows:

```
vector-def
    ::= VECTOR map [ = column ]
```

```
MODULE vectorexamples(NUMBER:v1:v2).
VECTOR v1 = 2.
VECTOR v2 = [2,3].
CLOSE.
```

Figure 12: Vector objects.

```
PRINT v1    PRINT 2*v2    PRINT v2*v2    PRINT pow(v2,2)
varn        varn          varn           varn
2           4             4              4
            6             6              9
                          9
   (a)          (b)           (c)            (d)
```

Figure 13: Queries on vectors.

The VECTOR command creates a new map object that is initialized with the column if present. The map name is declared in the module declaration and the type of the vector (NUMBER or STRING) is inferred from the declaration. A vector is internally compiled as a map with one attribute which serves as the return variable.

Vectors can be used in the same way as variables in programming languages. Consider the definitions in figure 12. The PRINT query (a) in figure 13 lists the value of vector v1. In (b) all values of v2 are multiplied by two. In query (c) v2 is multiplied by itself, yielding three different values (duplicate values are removed from the answer relation). Note that the answer relation in (c) is different from that of (d) because v2 has more than one value.

6.2 Arrays

In analogy to vectors arrays are persistent objects that reside in the relational database. In addition arrays can be indexed and for a particular valuation of the indexes, one or more values can be stored:

```
array-def
    ::= ARRAY map ( attribute-list ) [ = relation ]
```

The ARRAY command creates a new map object that is initialized with the relation if present. The attribute list is the list of indexes of the array. The domain of the return value of the array is determined by the domain of the map (declared in the

```
MODULE arrayexamples(NUMBER:a2:a3:a4:i:j,STRING:a1).
ARRAY a1(i) = [[1,red],[2,green],[2,yellow],[3,blue]].
ARRAY a2(i) = [[1,10],[2,20]].
ARRAY a3(i,j) = [[1,1,100],[1,2,200],[2,1,300],[2,2,400]].
ARRAY a4(i) = [[1,10],[33,20],[53,30]].
CLOSE.
```

Figure 14: Array objects.

```
PRINT   PRINT          PRINT          PRINT
a1(2)   a2(i)+a3(1,i)  a2(i)+a3(i,1)  a2(i)+a3(i,j)
vars    varn           varn           varn
green   110            110            110
yellow  220            320            210
                                      320
                                      420
  (a)         (b)            (c)            (d)
```

Figure 15: Queries on arrays.

module declaration). An array with n attributes is internally compiled as a map with n+1 attributes. The extra attribute is the return variable of the map. Some examples are given in figure 14. The query (a) in figure 15 lists two return values for the index 2. In (b) and (c) the index i occurring in the invocation of a2 and a3 limits the number of tuples in the answer relation. Query (d) lists four tuples because now all indexes are variables. Choosing all index variables different in (d) would give the full cartesian product (eight tuples).

An array object may have a non-dense index (i.e. not for all index values in the index range the array object needs to have a return value) as illustrated in the object a4. In this respect array objects behave like tables in the 'B' language (see [MEER81]). Also the type of an array index is not restricted to be numeric (as in the above examples) but string indexes are also allowed. Finally array objects may be used in the 'reverse' direction, so that for a given return value the associated index value(s) are calculated.

7 Clausal objects

Clauses are comparable to views in relational database languages ([STON75]). In the same way as a view serves as a macro facility with respect to a base table in a relational

```
CLAUSE true WHEN 0=0.        CLAUSE false WHEN 0=1.
SHOW true                    SHOW false
yes                          no
        (a)                          (b)
```

Figure 16: Defining true and false.

```
CLAUSE p(x) WHEN x=1.    CLAUSE p(x) WHEN
CLAUSE p(x) WHEN x=2.    x=1 OR x=2 OR x=3.
CLAUSE p(x) WHEN x=3.
        (a)                      (b)
```

Figure 17: Extending a clausal object in a single module.

database, clausal objects can be used as macro facilities for tabular objects. The syntax for clause definitions and invocations is as follows:

```
clause-def
    ::= CLAUSE clause [ ( attribute-list ) ] WHEN rule-exp
    |   CLAUSE clause ( * ) WHEN rule-exp
    |   CLAUSE clause ( / ) WHEN rule-exp
clause-invoc
    ::= clause [ ( arguments ) ]
```

In the first defining form the attribute list is optional. As a consequence the constants TRUE and FALSE can be defined as in figure 16. In the second (short) defining form, the attributes of the new clausal object are all the variables occurring in the rule expression. Since no specific order of attributes is enforced by the '(*)' notation, clauses defined by the second form can only be accessed in attribute notation. The third defining form can be used if a clausal object is defined with several clause definitions. In this case the clause definition has the same attributes as the definition that is already compiled for the object and the attributes need not be restated. Specifying a clausal object with several clause definitions expresses disjunction between the clause definitions. For this purpose also the first defining form can be used but then the attribute list is equal to the attribute list of the previously compiled definition. The definitions (a) and (b) in figure 17 for 'p' are equivalent.

The definition of a clausal (or functor) object may span *several* modules. Suppose a clause is defined by clause definitions in both module A and module B. If module A is imported in module B, the definition of the object is the disjunction of the clause

```
MODULE a(NUMBER:x).
CLAUSE p(x) WHEN x=1.
CLOSE.
MODULE b(USING:a, NUMBER:x).
CLAUSE p(/) WHEN x=2.
CLAUSE p(/) WHEN x=3.
CLOSE.
```

Figure 18: Defining a clausal object with two modules.

definitions in A and the definitions in B. For instance compiling the modules in figure 18 would leave the same definition for 'p' as in (a) and (b) of figure 17.

7.1 Domain objects

In creating modules it is sometimes useful to restrict the domain of a variable. This may be achieved by defining a domain object representing the restrictions for the domain. A domain object is any logical object which has only one attribute. For each occurrence of a variable in the rule expression, an invocation of such a domain object can be used to restrict the domain of the variable. To relief the user from repeatedly specifying domain restrictions in a rule expression, it is allowed to immediately declare the domain restrictions for a variable in the module declaration (see the section on module declarations above). For each restricted variable its domain object is compiled automatically into the rule expression. The type of the variable (NUMBER or STRING) is inferred from the domain of the (only) attribute in the domain object. Vectors and enumerations are appropriate to serve as a domain object since they are created as extensional objects with only one attribute.

As an example consider the declaration of the domains 'sizes' and 'colors' in figure 19. Due to the domain specification the compiler adds domain restrictions to the rule expression 'product(name,size,color)' in the definition of the clause 'restrictedproduct' so that the original clause is replaced by the following:

```
CLAUSE restrictedproduct(name,size,color) WHEN
    product(name,size,color) AND sizes(size) AND colors(color).
```

7.2 Declarative and conditional operators

Both rule expressions and constraints have their own operators for conjunction, disjunction and negation. The AND, OR and NOT operators are declarative operators used within the **RL/1** language in object definitions. The $\wedge$, $\vee$ and $\neg$ operators belong

```
MODULE domains(STRING:x:y).
CLAUSE sizes(x) WHEN x='small' ∨ x='medium' ∨ x='large'.
CLAUSE colors(x) WHEN x='red' ∨ x='green'.
CLOSE.
MODULE products(USING:domains, STRING:name,
       DOM sizes:size, DOM colors:color).
TABLE product(name,size,color)=
       [[chair,large,green],
       [mousemat,small,blue]].
CLAUSE restrictedproduct(name,size,color) WHEN
       product(name,size,color).
CLOSE.
```

Figure 19: Domain Objects.

to the database language **DL** and are intended to correspond to operators directly implemented by the database system. They are compiled as operations in the query to the underlying relational database. This may lead to the situation where, even though semantically both types of operators result in the same definition for an object, operationally there may be an important difference in the evaluation strategy. Declarative operators, contrary to conditional operators, are subject to expansion before they are executed on the database. To illustrate the expansion of declarative operators consider what happens if the declarative OR operator was used instead of ∨ for the definition of colors and sizes:

```
CLAUSE sizes(x) WHEN x='small' OR x='medium' OR x='large'.
CLAUSE colors(x) WHEN x='red' OR x='green'.
SHOW sizes(x) AND colors(y)
```

The answer relation for the query lists the cartesian product of sizes with colors (six tuples). However evaluation poses serious problems. To obtain the answer, the rule expression would first be expanded to:

```
x='small' AND y='red' OR ... OR x='large' AND y='green'
```

For the above example, expansion can be managed but as the number of color or size values grows, the size of the expanded rule expression increases very rapidly. In cases where the expansion was not intended, such as the above domain restrictions on sizes and colors, the conditional disjunction operator can be used to avoid expansion. As may be clear from the examples above, it is up to the user to choose between conditional or declarative operators, since he knows best how the operator will be used in rule expressions.

```
MODULE convs(NUMBER:x1:y1:x2:y2:xc:yc:wc:hc, STRING:name).
TABLE rectdata(name,x1,y1,x2,y2)=
        [[ra,0,0,4,4],
        [rb,2,2,7,7],
        [rc,6,3,9,6]].
CLAUSE conv(x1,y1,x2,y2,xc,yc,wc,hc) WHEN
        x1=xc-wc AND y1=yc-hc AND x2=xc+wc AND y2=yc+hc.
CLOSE.
```

Figure 20: A conversion rule.

7.3 Declarative use of logical objects

Because logical objects are declarative representations of knowledge it is not known beforehand how the variables in an object invocation will be used. In fact the logical object may be used in many different ways. For each of these applications the same object can be used. For instance suppose rectangles are stored in a table with four numeric attributes x1,y1,x2,y2, the pair (x1,y1) denoting the origin and (x2,y2) denoting the corner. A user might want a list of these rectangles in an alternative representation system with centre points (xc,yc) and size pairs (wc,hc) giving the distance between the centre and the origin. The alternative centre representation for rectangles could then be calculated with use of the clausal conversion rule given in figure 20.

The conversion of point representation ((x1,y1), (x2,y2)) to centre representation ((xc,yc), (wc,hc)) requires solving the constraints x1=xc-wc, y1=yc-hc, x2=xc+wc and y2=yc+hc for the variables xc, yc, wc and hc. The constraint solver takes care of the solving process:

```
SHOW rectdata(name,x1,y1,x2,y2) AND conv(x1,y1,x2,y2,xc,yc,wc,hc)
```

name	x1	y1	x2	y2	xc	yc	wc	hc
ra	0	0	4	4	2	2	2	2
rb	2	2	7	7	4.5	4.5	2.5	2.5
rc	6	3	9	6	7.5	4.5	1.5	1.5

Another user might want to use the 'conv' rule in a different way. Suppose he wants to know for a centre point (7.5,4.5) and a size pair (1.5,1.5) the associated rectangle in origin and corner representation. For this query the same clause can be used since the rule states declaratively what is known to be true about the representation systems:

```
SHOW conv(x1,y1,x2,y2,7.5,4.5,1.5,1.5)
```

x1	y1	x2	y2
6	3	9	6

A third user might want a representation of origins and centre points to be converted in a representation of corners and sizes and still the conversion rule can be used:

```
SHOW conv(6,3,x2,y2,7.5,4.5,wc,hc)
 x2  y2  wc   hc
 9   6   1.5  1.5
```

Other applications include overspecification so that inconsistencies can be inferred. The resulting relation is empty:

```
SHOW conv(6,3,9,6,0,0,wc,hc)
 wc     hc
 void
```

Also underspecification can occur in which case the answer relation is infinite and not presented to the user:

```
SHOW conv(6,3,x2,y2,xc,yc,wc,hc)
 error
```

8 Functor objects

Functors add new functions to the set of standard functions already available in the system. Functor objects are defined with use of rule expressions and as a consequence they may involve invocations to other logical objects. The syntax for functor definitions and invocations is as follows:

```
functor-def
    ::= FUNCTOR functor ( attribute-list ) WHEN rule-exp
    |   FUNCTOR functor ( * , variable ) WHEN rule-exp
    |   FUNCTOR functor ( / ) WHEN rule-exp
functor-invoc
    ::= functor [ ( arguments ) ]
```

In the first defining form the last attribute of the attribute list is the return variable of the functor and the other attributes represent the arguments of the functor. Functor definitions that have only one attribute (i.e. the return variable) are allowed. In the second (short) defining form, 'variable' is the return variable of the functor. All variables occurring in the rule expression, besides the return variable, become attributes of the newly defined functor. Since no specific order of attributes is enforced by the '(*,variable)' notation, functors defined in this way can only be accessed in attribute notation.

```
FUNCTOR int(x,y) WHEN y=integer(x).
FUNCTOR sum(x,y,z) WHEN z=x+y.
FUNCTOR abs(x,y) WHEN y=x AND x≥0 OR y=-x AND x<0.
SHOW x=int(3.5).  % yields 3
SHOW x=sum(sum(1,2),3).  % yields 6
SHOW x=abs(-10).  % yields 10
```

Figure 21: Functor objects.

```
MODULE constantexamples(NUMBER:x,STRING:s).
FUNCTOR pi(x) WHEN x=3.141.
FUNCTOR ph(s) WHEN s='phone'.
CONST pi1 = 3.141.
CONST ph1 = 'phone'.
CONST e = exp(1).
SHOW s = ph CAT 'net'.  % yields 'phonenet'.
SHOW x = pi+5.  % yields 8.141
CLOSE.
```

Figure 22: Constant objects.

The third defining form can be used in case a functor object is defined by several functor definitions or the definition spans more than one module. As for clauses the first defining form can also be used for extending a functor object. Functors can be used as a macro facility for standard functions in the same way as clauses serve as macro facility for tables. For instance if the user wants to abbreviate the name of a standard function 'integer' a new functor can be declared as in figure 21.

8.1 Constants

Since in a functor invocation the argument list is optional, functors can be used for the definition of constants. The defined constant is then used in the same way as other numerical or string constants (see figure 22). For user convenience we allow constants to be defined in a short format:

```
const-def
    ::= CONST functor = scalar-exp
```

```
MODULE quads(NUMBER:a:b:c:d:e:x).
TABLE data(a,b,c)=[[1,8,7],[1,4,0]].
FUNCTOR quad(a,b,c,x) WHEN
        x=(-b+e)/(2*a) AND d=b*b-4*a*c AND
        ( e=sqrt(d) AND d≥0 OR e=-sqrt(d) AND d>0 ).
CLOSE.
```

Figure 23: A multivalued functor.

The type of the constant is not declared in the module declaration but is instead inferred from the type of the scalar expression. The constant definition may contain invocations to arbitrary functional objects.

8.2 Multivalued functors

The full expressiveness of rule expressions is available for functors, so functors are not restricted to yield only one function value for an assignment of argument values. For example a functor yielding a value x for given values of a, b and c such that $a * x^2 + b * x + c = 0$ could be defined as in figure 23.

The OR operator in the functor definition results in two x values for each assignment of values to variables a,b and c, since the conditions d≥0 and d>0 are not exclusive:

```
SHOW data(a,b,c) AND x=quad(a,b,c)
```

x	a	b	c
0	1	4	0
-1	1	8	7
-4	1	4	0
-7	1	8	7

8.3 Type hierarchies

If types are interpreted as subsets of a value domain, it is quite natural to establish a partial ordering among types based on set inclusion: this is the key idea underlying type hierarchies (see [ALB88] and [MIT88]). As a consequence a type t is a subtype of another type u when all the values of t are also values of u. Domain objects can represent such type hierarchies. Figure 24 declares a (rather incomplete) type hierarchy of animals with use of vectors and functors as domain objects.

The functional objects bird and fish represent sets of animals and are subtypes of vertebrate. Vertebrate itself is a subtype of animal. Query (a) below shows that indeed a duck is warm blooded. Query (b) indicates that there are no warm blooded fish. The

```
MODULE animaltypes(STRING:x
       herbivore:carnivore:bird:fish:invertebrate).
VECTOR herbivore= [deer,elephant,kangaroo].
VECTOR carnivore= [dog,bat,cat].
VECTOR bird= [duck,eagle,ostrich].
VECTOR fish= [shark,dogfish,plaice].
VECTOR invertebrate= [hydra,sponge,starfish].
FUNCTOR mammal(x) WHEN x=carnivore OR x=herbivore.
FUNCTOR vertebrate(x) WHEN x=mammal OR x=bird OR x=fish.
FUNCTOR animal(x) WHEN x=vertebrate OR x=invertebrate.
CLOSE.
MODULE animals(USING:animaltypes, STRING:x).
CLAUSE warm_blooded(x) WHEN x=mammal.
CLAUSE warm_blooded(x) WHEN x=bird.
CLAUSE cold_blooded(x) WHEN x=fish.
CLOSE.
```

Figure 24: Functional objects in a type hierarchy.

same question for birds in (c) is answered positively because there are warm blooded birds.

```
SHOW                    SHOW                    SHOW
warm_blooded('duck')    warm_blooded(fish)      warm_blooded(bird)
yes                     no                      yes
        (a)                     (b)                     (c)
```

Functional domain objects are allowed at the same locations as normal constants. So the question whether there are animals that are both a mammal and a bird can be stated as in (a). The same question for mammals and carnivores is represented as in (b). The query (c) lists precisely the animals that are in the intersection:

```
SHOW mammal=bird  SHOW mammal=carnivore  SHOW x=mammal AND x=carnivore
no                yes                    x
                                         dog
                                         bat
                                         cat
      (a)                 (b)                     (c)
```

The TREE command (described in the section on request commands below) lists all the objects used in the definition of an object so it can applied to give the subtypes of 'animal':

```
TREE animal.
animal
    vertebrate
        mammal
            carnivore
            herbivore
        bird
        fish
    invertebrate
```

9 Commands

The rule language consists of a sequence of commands. A command is one of the following items followed by a period (queries and requests are described in forthcoming sections):

```
command
    ::= { table-def | map-def | clause-def | functor-def
    |   enum-def | map-def | vector-def | array-def | const-def
    |   query | request | module-decl | module-term } .
```

9.1 Query commands

All query commands in **RL/1** result in an answer relation. An answer relation consists of attributes and a (possibly empty) set of tuples. Optionally the result of a query command can be stored in a table object. The table object is then created with the attributes of the answer relation. There are the following types of query commands:

```
query
    ::= INFER ( attribute-list ) WHEN rule-exp [ TO table ]
    |   SHOW rule-exp [ TO table ]
    |   LIST object [ TO table ]
    |   PRINT scalar-exp [ TO table ]
```

The INFER command is the most general command. It yields an answer relation with attributes equal to the attribute list between the INFER and WHEN keywords. The SHOW command yields an answer relation whose attributes are the variables occurring in the rule expression. The command is useful as a shorthand for the INFER command if all variables of the rule expression are requested.

The LIST command applied to a logical object yields an answer relation corresponding to the denoted relation of the object. The command is useful for instance, if queries

are saved as named clauses for future use. Listing the clause name, then suffices to evaluate the stored query. A LIST command applied to a module lists all objects defined in the module.

Evaluation of scalar expressions is achieved with the PRINT query and this query also yields an answer relation. The answer relation has one attribute with a system generated name starting with the letters 'var'. The answer relation may contain more than one value for the single attribute. In order for the system generated variables not to clash with normal variables, in **RL/1** variables (or attributes) are not allowed to start with the letters 'var'.

As a special case an answer relation with no attributes but with one tuple being the empty set, represents the answer 'yes'. An answer relation with no attributes and an empty set of tuples represents the answer 'no'. Having the answer relations 'yes' and 'no' is useful for queries without variables that are true or false (see (a) and (b) below). If the answer relation does have attributes but the tuple set is empty then the tuple set is denoted by 'void' (see (c) and (d)).

```
SHOW 4>3   SHOW 4<3   SHOW x=1   SHOW x=1 AND x<1
yes        no         x          x
                      1          void
  (a)        (b)        (c)          (d)
```

9.2 Request commands

In this section we list some commands that operate on objects. These commands are defined for all the object types. Currently the next requests are implemented:

```
request
        ::= HELP object
        |   DROP object
        |   KILL object
        |   TREE object
        |   USED object
```

The HELP command gives more specific information about an object. For each attribute of the object, the attribute name and the attribute domain are shown. Also the names of the modules where the object is defined are listed together with the defining text. The help command is particularly useful to locate the definition of an object if more modules are compiled.

The DROP command erases all information about an object. This can be useful in case an object is no longer needed and occupies storage. If the command is applied on a logical object, the definition of the object is destroyed. For extensional objects also the extension in the underlying relational database is discarded. The drop command is canceled if there are other logical objects that use the logical object in their definition.

Applied on a module object, the command drops all objects defined inside the module and the module declaration itself. The drop command is canceled if there are other modules using the module.

A KILL command is similar to a drop command except that it is never canceled. Instead objects that directly or indirectly use the object to be killed, are also killed themselves. The command avoids discarding each object individually as might be needed when using the drop command.

For a particular object the TREE command generates a tree of dependencies. For logical objects the command lists the objects that are used in the definition of the logical object (either directly or indirectly). For a module object, all modules that are imported directly or indirectly are listed. To find out what objects use a particular object the USED command can be applied. For a logical object all objects are listed that invoke the object in their definition. If the command is applied on a module all modules that directly import the module with a using declaration are enumerated.

10 Conclusion

In this report the syntax of the language **RL/1** was described. We have shown with examples how logic programming, relational databases and constraint solving can be integrated into a single relational framework. The language **RL/1** has been implemented in PROLOG and runs on a SUN SPARCstation 1. The examples in this report that are shown with an answer relation were executed on this implementation.

References

[ALB88] Albano, A., Giannotti, F., Orsini,R., Pedreschi,D., *The Type System of Galileo,* Data Types and Persistence, (Eds. Atkinson, Buneman, Morrison), Springer-Verlag 1988, 102-119.

[CLOCK81] Clocksin, W.F. & Mellish, C.S., *Programming in Prolog,* Springer-Verlag, 1981.

[DATE87] Date, C.J., *A Guide to the SQL Standard,* Addison-Wesley Publishing Company 1987.

[DATE89] Date, C.J. & White, C.J., *A Guide to DB2,* (Third Edition), Addison-Wesley Publishing Company 1989.

[DEN88a] van Denneheuvel, S. & van Emde Boas, P., *Constraint solving for databases,* Proc. of NAIC **1**, Apr. 1988

[DEN88b] van Denneheuvel, S. & van Emde Boas, P., *Towards implementing RL,* Preprint CT-88-11, Institute for Language, Logic and Information, University of Amsterdam, 1988

[DEN89] van Denneheuvel, S. & Renardel de Lavalette, G. R., *Normalisation of Database expressions involving Calculations,* Logic Group Preprint Series No.45, Department of Philosophy, University of Utrecht, 1989

[GOLD81] Goldberg, A. & Robson, D., *Smalltalk-80:The Language and its Implementation,* Addison-Wesly, 1983

[HANS88] Hansen, M.R., *Algebraic Optimization of Recursive Database Queries,* Information Systems and Operations Research 26 (1988) 286-298

[HANS89] Hansen, M.R., Hansen, B.S., Lucas, P. & van Emde Boas, P., *Integrating Relational Databases and Constraint Languages,* in Comput. Lang. Vol. **14**, No. 2, 63-82, 1989.

[MEER81] Meertens, L., *Draft proposal for the B programming language, MC* Series, printed at the Mathematical Centre, Amsterdam, 1981

[MIT88] Mitschang B., *Towards a unified view of design data and knowledge representation,* Proc. from the second Int. Conf. of Expert Database Systems (Ed. L. Kerschberg) (1988) 133-159

[NAQV89] Naqvi, S. & Tsur, S., *A Logical Language for Data and Knowledge bases,* Computer Science Press, 1989.

[ROES87] Roessingh, M.J., *RL losgelaten op aanwijzing 111,* Rep. FVI-UvA 87-18, Dec. 1987.

[STON75] Stonebraker, M., *Implementation of Integrity Constraints and Views by Query Modification,* Proc. 1975 ACM-SIGMOD Conference, San Jose, Ca., May 1975.

[ULL89] Ullman, J.D., *Principles of Data and Knowledge - Base Systems,* Volume II: The New Technologies, Computer Science Press, 1989.

[VEMD86a] van Emde Boas, P., *RL, a Language for Enhanced Rule Bases Database Processing,* Working Document, Rep IBM Research, RJ 4869 (51299)

[VEMD86b] van Emde Boas, P., *A semantical model for the integration and modularization of rules,* Proceedings MFCS 12, Bratislava, August 1986, Springer Lecture Notes in Computer Science **233** (1986), 78-92

[VEMD86c] van Emde Boas, H. & van Emde Boas, P., *Storing and Evaluating Horn-Clause Rules in a Relational Database,* IBM J. Res. Develop. **30** (1), (1986), 80-92

The ITLI Prepublication Series

1986

86-01 The Institute of Language, Logic and Information
86-02 Peter van Emde Boas — A Semantical Model for Integration and Modularization of Rules
86-03 Johan van Benthem — Categorial Grammar and Lambda Calculus
86-04 Reinhard Muskens — A Relational Formulation of the Theory of Types
86-05 Kenneth A. Bowen, Dick de Jongh — Some Complete Logics for Branched Time, Part I Well-founded Time,
86-06 Johan van Benthem — Logical Syntax Forward looking Operators

1987

87-01 Jeroen Groenendijk, Martin Stokhof — Type shifting Rules and the Semantics of Interrogatives
87-02 Renate Bartsch — Frame Representations and Discourse Representations
87-03 Jan Willem Klop, Roel de Vrijer — Unique Normal Forms for Lambda Calculus with Surjective Pairing
87-04 Johan van Benthem — Polyadic quantifiers
87-05 Víctor Sánchez Valencia — Traditional Logicians and de Morgan's Example
87-06 Eleonore Oversteegen — Temporal Adverbials in the Two Track Theory of Time
87-07 Johan van Benthem — Categorial Grammar and Type Theory
87-08 Renate Bartsch — The Construction of Properties under Perspectives
87-09 Herman Hendriks — Type Change in Semantics: The Scope of Quantification and Coordination

1988

Logic, Semantics and Philosophy of Language:

LP-88-01 Michiel van Lambalgen — Algorithmic Information Theory
LP-88-02 Yde Venema — Expressiveness and Completeness of an Interval Tense Logic
LP-88-03 — Year Report 1987
LP-88-04 Reinhard Muskens — Going partial in Montague Grammar
LP-88-05 Johan van Benthem — Logical Constants across Varying Types
LP-88-06 Johan van Benthem — Semantic Parallels in Natural Language and Computation
LP-88-07 Renate Bartsch — Tenses, Aspects, and their Scopes in Discourse
LP-88-08 Jeroen Groenendijk, Martin Stokhof — Context and Information in Dynamic Semantics
LP-88-09 Theo M.V. Janssen — A mathematical model for the CAT framework of Eurotra
LP-88-10 Anneke Kleppe — A Blissymbolics Translation Program

Mathematical Logic and Foundations:

ML-88-01 Jaap van Oosten — Lifschitz' Realizabiility
ML-88-02 M.D.G. Swaen — The Arithmetical Fragment of Martin Löf's Type Theories with weak Σ-elimination
ML-88-03 Dick de Jongh, Frank Veltman — Provability Logics for Relative Interpretability
ML-88-04 A.S. Troelstra — On the Early History of Intuitionistic Logic
ML-88-05 A.S. Troelstra — Remarks on Intuitionism and the Philosophy of Mathematics

Computation and Complexity Theory:

CT-88-01 Ming Li, Paul M.B.Vitanyi — Two Decades of Applied Kolmogorov Complexity
CT-88-02 Michiel H.M. Smid — General Lower Bounds for the Partitioning of Range Trees
CT-88-03 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas — Maintaining Multiple Representations of Dynamic Data Structures
CT-88-04 Dick de Jongh, Lex Hendriks Gerard R. Renardel de Lavalette — Computations in Fragments of Intuitionistic Propositional Logic
CT-88-05 Peter van Emde Boas — Machine Models and Simulations (revised version)
CT-88-06 Michiel H.M. Smid — A Data Structure for the Union-find Problem having good Single-Operation Complexity
CT-88-07 Johan van Benthem — Time, Logic and Computation
CT-88-08 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas — Multiple Representations of Dynamic Data Structures
CT-88-09 Theo M.V. Janssen — Towards a Universal Parsing Algorithm for Functional Grammar
CT-88-10 Edith Spaan, Leen Torenvliet, Peter van Emde Boas — Nondeterminism, Fairness and a Fundamental Analogy
CT-88-11 Sieger van Denneheuvel, Peter van Emde Boas — Towards implementing RL

Other prepublications:

X-88-01 Marc Jumelet — On Solovay's Completeness Theorem

1989

Logic, Semantics and Philosophy of Language:

LP-89-01 Johan van Benthem — The Fine-Structure of Categorial Semantics
LP-89-02 Jeroen Groenendijk, Martin Stokhof — Dynamic Predicate Logic, towards a compositional, non-representational semantics of discourse
LP-89-03 Yde Venema — Two-dimensional Modal Logics for Relation Algebras and Temporal Logic of Intervals
LP-89-04 Johan van Benthem — Language in Action
LP-89-05 Johan van Benthem — Modal Logic as a Theory of Information
LP-89-06 Andreja Prijatelj — Intensional Lambek Calculi: Theory and Application
LP-89-07 Heinrich Wansing — The Adequacy Problem for Sequential Propositional Logic
LP-89-08 Víctor Sánchez Valencia — Peirce's Propositional Logic: From Algebra to Graphs

Mathematical Logic and Foundations:

ML-89-01 Dick de Jongh, Albert Visser — Explicit Fixed Points for Interpretability Logic
ML-89-02 Roel de Vrijer — Extending the Lambda Calculus with Surjective Pairing is conservative
ML-89-03 Dick de Jongh, Franco Montagna — Rosser Orderings and Free Variables
ML-89-04 Dick de Jongh, Marc Jumelet, Franco Montagna — On the Proof of Solovay's Theorem
ML-89-05 Rineke Verbrugge — Σ-completeness and Bounded Arithmetic
ML-89-06 Michiel van Lambalgen — The Axiomatization of Randomness
ML-89-07 Dirk Roorda — Elementary Inductive Definitions in HA: from Strictly Positive towards Monotone
ML-89-08 Dirk Roorda — Investigations into Classical Linear Logic
ML-89-09 Alessandra Carbone — Provable Fixed points in $I\Delta_0+\Omega_1$

Computation and Complexity Theory:

CT-89-01 Michiel H.M. Smid — Dynamic Deferred Data Structures
CT-89-02 Peter van Emde Boas — Machine Models and Simulations
CT-89-03 Ming Li, Herman Neuféglise, Leen Torenvliet, Peter van Emde Boas — On Space Efficient Simulations
CT-89-04 Harry Buhrman, Leen Torenvliet — A Comparison of Reductions on Nondeterministic Space
CT-89-05 Pieter H. Hartel, Michiel H.M. Smid Leen Torenvliet, Willem G. Vree — A Parallel Functional Implementation of Range Queries
CT-89-06 H.W. Lenstra, Jr. — Finding Isomorphisms between Finite Fields
CT-89-07 Ming Li, Paul M.B. Vitanyi — A Theory of Learning Simple Concepts under Simple Distributions and Average Case Complexity for the Universal Distribution (Prel. Version)
CT-89-08 Harry Buhrman, Steven Homer Leen Torenvliet — Honest Reductions, Completeness and Nondeterminstic Complexity Classes
CT-89-09 Harry Buhrman, Edith Spaan, Leen Torenvliet — On Adaptive Resource Bounded Computations
CT-89-10 Sieger van Denneheuvel — The Rule Language RL/1

Other Prepublications:

X-89-01 Marianne Kalsbeek — An Orey Sentence for Predicative Arithmetic
X-89-02 G. Wagemakers — New Foundations: a Survey of Quine's Set Theory
X-89-03 A.S. Troelstra — Index of the Heyting Nachlass
X-89-04 Jeroen Groenendijk, Martin Stokhof — Dynamic Montague Grammar, a first sketch
X-89-05 Maarten de Rijke — The Modal Theory of Inequality