Institute for Language, Logic and Information

AN OVERVIEW OF THE RULE LANGUAGE RL/1

Sieger van Denneheuvel
Peter van Emde Boas

ITLI Prepublication Series
X-90-10

University of Amsterdam

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, Forward looking Operators
86-06	Johan van Benthem	Logical Syntax
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
LP-89-09	Zhisheng Huang	Dependency of Belief in Distributed Systems
		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
CT-89-11	Zhisheng Huang, Sieger van Denneheuvel Peter van Emde Boas	Towards Functional Classification of Recursive Query Processing
		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
X-89-06	Peter van Emde Boas	Een Relationele Semantiek voor Conceptueel Modelleren: Het RL-project
1990	SEE INSIDE BACK COVER	

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

AN OVERVIEW OF THE RULE LANGUAGE RL/1

Sieger van Denneheuvel
Peter van Emde Boas
Department of Mathematics and Computer Science
University of Amsterdam

Received July 1990

An overview of the Rule Language RL/1

Sieger van Denneheuvel & Peter van Emde Boas*
Department of Mathematics and Computer Science, University of Amsterdam
Plantage Muidergracht 24, 1018 TV Amsterdam

Abstract

In this paper we introduce and illustrate the rule language **RL/1**. This language was designed as an intermediate step towards implementing the more extensive rule language **RL** intended to become a tool for integrating logical and functional programming and constraint solving with relational databases into a relational framework.

1 Introduction

The language **RL/1** described in this paper is intended to be an illustrative prototype and an intermediate step towards implementing the declarative rule language **RL** (see [7], [8] and [9]). The language represents an effort to integrate logical and functional programming with constraint solving and relational databases. Towards the user, the free use of equational constraints together with the ease of logic programming constitutes a declarative and user friendly framework for expressing a knowledge base. On the other hand query processing should be executed with help of an existing relational database system; knowledge and queries expressed in **RL/1** are to be preprocessed by a constraint solver, and to be compiled into a database query language in order that large amounts of data can be processed effectively. In the current prototype **RL/1** we do not focus on recursion as is done in the NAIL! system [6] and the **LDL** logical database [5] but rather on the integration of a subsystem solving (numeric) constraints with a relational database system; the architecture for such an integrated system has been presented elsewhere [2].

This paper intends to illustrate the user perspective of our intended system by providing some illustrative syntax and semantics for the prototype language **RL/1** and some examples of **RL/1** modules. A compiler for the language **RL/1** has been implemented in a prototype system. All examples presented in this paper were processed successfully by this compiler in combination with a constraint solver and a relational database coded directly in **Prolog**. A version of the compiler producing standard **SQL** is currently being developed.

2 Logical objects in RL/1

In the **RL** language design a program consists of modules, each module describing a system of relations. Relations are described using expressions originating from the worlds of relational databases, logic programming and equational logic. Atomic relations can be combined using operators originating from these worlds, providing maximal freedom and full conceptual transparence to the user. Syntactic categories are introduced to keep track of the origin and nature of the various relational expressions. A query in the **RL** language consists of a relational expression which is to be evaluated in the context of an existing program, the result of which is shown to the user.

The **RL/1** language was designed to become an implemented prototype for the full **RL** language, which eliminates some amount of syntactic and semantic complexity while preserving the main ideas of integrating relational databases with logic programming and algebraic constraint solving.

In **RL/1** relations, called 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. From the logical objects tables and maps are *extensional objects* in the sense that the relation denoted by the object is

*also CWI-AP6, Amsterdam

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.

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 (undirected) or functional (directed). Tables and clauses are *relational objects* and therefore comparable to tables and views of a relational database. Maps and functors on the other hand are *functional objects*, where conceptually one attribute is considered to be dependent of the others (where dependence doesn't exclude that the function may be multivalued). They are invoked in scalar expressions, whereas relational objects are used in rule expressions. The difference between functional and relational objects therefore is primarily a matter of syntax; the first kind is used in functional notation and the latter is invoked as a predicate. For evaluation of rule expressions the SHOW query is available and for scalar expressions the PRINT query. The above properties are summarized in Figure 1.

3 The data language DL

Each language in the **RL** family contains a sublanguage **DL** called the *data language* which is intended to reflect the nature of the underlying database system. In this sublanguage values and relations are expressed which are intended to be evaluated directly by this database. The operators from the data language **DL** need therefore to be supported by the corresponding relational database language. On the other hand the language **DL** should be rich enough so that features which are provided by the database can be accessed from the **RL/1** system directly.

Given the fact that in our **RL/1** prototype the database is implemented in **Prolog** an ad-hoc language **DL** has been designed for **RL/1**. This language **DL** has two scalar types, namely NUMBER with constants from category 'num-val' and STRING with constants from 'str-val'. The syntax of scalar expressions and constraints is defined in Figure 2 (predicates are not listed). In the grammar 'functor-inv' and 'map-inv' are invocations to functional objects which are described in the sections below. The grammar is incomplete with respect to the categories 'str-func' and 'num-func'. In our grammar notation below 'x-list' is a category consisting of a comma separated list of one or more 'x's. Similarly 'x-colonlist' is a category consisting of a colon separated list of 'x's.

4 The rule language RL/1

In contrast with the data language the rule language is independent of the underlying relational database. The rule language has a declarative nature providing various means to express what is known about a domain of interest. It does *not* specify how the knowledge is to be used for evaluation of a particular query. The compiler for the rule language will compile this query in the context of the **RL/1** program into a program which can be optimized and evaluated by the database.

As indicated before there are extensional and intensional objects. In **RL/1** an extensional object can be initialized by an explicit listing of its value when it is first created. This value consists of a set of tuples. 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. A 'column' is a set of tuples consisting of one value only. All values in a column are of the same type (either NUMBER or STRING). In case a relation or a column has only one tuple or value a pair of square brackets may be omitted. Both relations and columns can be declared external indicating that the extension of the object is already stored in the database. The syntax of extensions is given in Figure 3.

```
num-fac
  ::= [ - ] { num-val | functor-inv }
  |    [ - ] { map-inv | num-func }
  |    [ - ] { variable | ( num-exp ) }
num-term
  ::= num-fac | num-term { * | / } num-fac
num-exp
  ::= num-term | num-exp { + | - } num-term
str-term
  ::= functor-inv | map-inv
  |    str-func | variable | str-val
str-exp ::= str-term | str-exp CAT str-term
scalar-exp ::= num-exp | str-exp

comp ::= { = | < | > | ≤ | ≥ | ≠ }
constraint-fac
  ::= [ ¬ ] { predicate | ( constraint ) }
  |    [ ¬ ] num-exp comp num-exp
  |    [ ¬ ] str-exp comp str-exp
constraint-term
  ::= constraint-fac
  |    constraint-term ∧ constraint-fac
constraint
  ::= constraint-term
  |    constraint ∨ constraint-term
```

Figure 2: Scalar expressions and constraints.

```
val ::= num-val | str-val | NULL
row ::= "[" val-list "]"
column
  ::= "[" val-list "]" | val | EXTERN
relation
  ::= "[" row-list "]" | row | EXTERN
```

Figure 3: Object extensions.

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

Figure 4: Rule expressions.

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

Figure 5: Grammar for arguments.

	Positional	Attributive	Relation
(a)	t(*)	t([a,b,c])	$\underline{\text{a b c}}$ 1 2 3
(b)	t(x,y,#)	t([a=x,b=y])	$\underline{\text{x y}}$ 1 2
(c)	t(x+1,y,#)	t([a=x+1,b=y])	$\underline{\text{x y}}$ 0 2
(d)	t(3,y,#)	t([a=3,b=y])	$\underline{\text{y}}$ void

Figure 6: Positional and attribute notation.

In **RL/1** an intensional object is defined with use of the rule expressions in Figure 4 (the categories 'table-inv' and 'clause-inv' are described below). The declarative AND and OR operators used in rule expressions are more general than the corresponding operators join and union in relational algebra (see also [3]) since their operands are not restricted to be finite relations as is the case in relational algebra; instead these operands may be constraints or clause invocations that, by themselves, can represent intensional objects and thus potentially infinite relations. The question whether a rule expression as a whole represents a finite or infinite relation is in general undecidable; partial information about this question is obtained with help of a constraint solver (see [2]).

4.1 Object invocations

Both *positional notation* and *attribute notation* (Figure 5) can be used to supply arguments for invocations of extensional and intensional objects (cf. the language **LDL** in [5] where attribute notation can not be applied on intensional objects). This enables the user to use positional notation for objects with few attributes whereas for objects with many attributes, attribute notation may be preferred. In positional notation the positions of the invocation arguments should match the attribute positions in the object definition. 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.

In both notations substitutions of general scalar expressions for object attributes are allowed. In Figure 6 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.

4.2 Module definitions

In **RL/1** each identifier appearing in the module is declared either to denote a value of the types NUMBER or STRING, or to represent a logical object (see Figure 7). In the last case the type is determined by this logical object (see the section on domain objects below). In addition to a type declaration, a variable can have one or more property declarations. The property declarations listed in the grammar 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. For encapsulation

```
domain-decl
  ::= NUMBER : variable-colonlist
  |    STRING : variable-colonlist
  |    logical-obj :  variable-colonlist
property-decl
  ::= KEY : variable-colonlist
  |    NOTNULL : variable-colonlist
  |    PRIVATE : logical-obj-colonlist
  |    USING : module-colonlist
decl ::= domain-decl | property-decl
module-decl
  ::= MODULE module ( decl-list )
```

Figure 7: Module declarations.

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.

5 Tabular and map objects

Tabular and map objects are extensional objects corresponding 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 object is initialized with a relation; in case the relation is omitted the object is left empty. A table (or map) can be declared as an external relation by using the EXTERN option so that the object may already be filled with a large number of tuples:

```
table-def ::=
  TABLE table ( attr-list ) [ = relation ]
table-inv ::= table ( arguments )
map-def ::=
  MAP map ( attr-list ) [ = relation ]
map-inv ::= map [ ( arguments ) ]
```

The MAP command creates a map object with attributes in the order of the attribute list. The last attribute of the 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 8 lists some examples of maps (the PRINT query evaluates scalar expressions). Both tabular and map objects are defined with a *single* definition.

5.1 Vectors and arrays

Vectors are maps consisting of one column only; moreover this column has no attribute name. The syntax is as follows:

```
VECTOR map [ = column ]
```

The vector name is declared in the module declaration together with the type of the vector (NUMBER or STRING). 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 of vectors in Figure 8. The PRINT query (a) 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 itself has more than one value.

In analogy to vectors arrays are maps where the last attribute name is omitted.

```
ARRAY map ( attr-list ) [ = relation ]
```

```
MODULE maps(NUMBER:x:v1:v2,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.
VECTOR v1 = 2.  VECTOR v2 = [2,3].
PRINT map1(1).  % yields red
PRINT map1(2).  % yields green,yellow
```

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

Figure 8: Map and vector objects.

The type of the return value of the array is determined by the type listed for this array in the 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 9. The query (a) 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 [4]). Also the type of an array index is not restricted to be numeric (as in the above examples) but string indexes are also allowed.

6 Clausal objects

Clausal objects are defined with one or more clause rules:

```
clause-rule
  ::= CLAUSE clause [ ( attr-list ) ]
        WHEN rule-exp
  |   CLAUSE clause ( * ) WHEN rule-exp
  |   CLAUSE clause ( / ) WHEN rule-exp
clause-inv ::= clause [ ( arguments ) ]
```

In the first defining form the attribute may be omitted when empty; consequently the constants TRUE and FALSE can be defined as clausal objects. 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 by multiple clause rules. In this case the clause rule has the same attributes as the rule that was previously compiled for the object and the attributes need not be restated. Specifying a clausal object with several clause rules expresses *disjunction* between the clause rules. Using the OR operator, the definitions (a) and (b) in Figure 10 for 'p' are equivalent.

```
MODULE arrays(NUMBER:i:j,
  NUMBER:a2:a3:a4, 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]].
```

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

Figure 9: Arrays

```
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 10: Extending a clausal object in a module.

```
MODULE domains(STRING:x:y).
CLAUSE sizes(x) WHEN
  x='small' ∨ x='medium' ∨ x='large'.
CLAUSE colors(x) WHEN
  x='red' ∨ x='green'.
MODULE products(USING:domains,
  STRING:name, sizes:size, colors:color).
TABLE prod(name,size,color).
CLAUSE p(name,size,color) WHEN prod(*).
```

Figure 11: Domain Objects.

The definition of a clausal (or functor) object may span *several* modules. Suppose a clause is defined by clause rules 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 rules in A and the rules in B.

6.1 Domain objects

In creating modules it is sometimes useful to introduce a subtype by restricting the domain of a variable. In **RL/1** this is achieved by defining a domain object representing the restrictions for the domain. A domain object is a 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. However, in order 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 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 11. Due to the domain specification the compiler adds domain restrictions to the rule expression 'prod(name,size,color)' in the definition of the clause 'p' so that the original clause is replaced by the following:

```
CLAUSE p(name,size,color) WHEN
  prod(name,size,color)
  AND sizes(size) AND colors(color).
```

7 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:

```
functor-rule
  ::= FUNCTOR functor ( attr-list )
          WHEN rule-exp
    |   FUNCTOR functor ( * , variable )
          WHEN rule-exp
    |   FUNCTOR functor ( / ) WHEN rule-exp
functor-inv ::= 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. The third defining form can be used in case a functor

```
MODULE functors(NUMBER:a:b:c:d:e:x:y:z).
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.
FUNCTOR pi(x) WHEN x=3.141.
SHOW x=sum(sum(1,2),3).  % yields 6
SHOW x=abs(-10).  % yields 10
SHOW x=pi+5.  % yields 8.141
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 ).
```

Figure 12: Functor objects.

object is defined by several functor rules or the definition spans more than one module. As for clauses the first defining form can also be used for extending a functor object. Some examples are given in Figure 12.

The full expressiveness of rule expressions is available for functors also, 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 $a * x^2 + b * x + c = 0$ can be defined as in Figure 12. 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 $d \geq 0$ 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 Examples

Logical objects are declarative representations of knowledge; therefore it is not known beforehand how the attributes of an object are invoked. A single logical object may in fact be used in many different ways.

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 center points (xc,yc) and size pairs (wc,hc) giving the distance between the center and the origin. The alternative center representation for rectangles could then be calculated with use of the clausal conversion rule given in Figure 13.

The conversion of point representation ((x1,y1), (x2,y2)) to center 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(#,x1,y1,x2,y2)
  AND conv(x1,y1,x2,y2,xc,yc,wc,hc)
 x1  y1  x2  y2  xc   yc   wc   hc
 0   0   4   4   2    2    2    2
 2   2   7   7   4.5  4.5  2.5  2.5
 6   3   9   6   7.5  4.5  1.5  1.5
```

Another user might want to know for a center point (7.5,4.5) and a size pair (1.5,1.5) the associated

```
MODULE convs(NUMBER:x1:y1:x2:y2,
  NUMBER: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.
```

Figure 13: A conversion rule.

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 second example: one of the most spectacular events in astronomy is the passage of a comet. The orbit of a comet is usually available to observing astronomers in the form of tables listing the right ascention (RA for short) and the declination. These two dimensions together constitute the location of the comet in the sky. A problem with this tabular description of orbits is that for most comets the RA and declination are not given for each day during the period of visibility. It is up to the astronomer to compute the intermediate right ascentions and declinations for the missing days. What makes the problem nasty is that the RA's are given in a h-m-s notation and consequently need to be converted to seconds. After computation of the interval values has finished the seconds are converted back to h-m-s notation so that the telescope can be directed.

In the case of the comet Austin, which was visible in May 1990 in Holland, a table was available with positions given for each five days. The module in Figure 14 lists an application to compute the missing days. A clausal object 'interval' creates from the table 'cometdata' a new table with an RA and associated next RA in one tuple. The table 'intervaldays' is used to vary over the five days of the interval. Finally the 'ra' object lists the requested RA in h-m-s notation together with the number of the observation day:

```
list ra
 d   h   m   s
 0   1   21  2
 1   1   16  27
 2   1   11  52
 ⋮   ⋮   ⋮   ⋮
 14  0   12  25
```

An unexpected application is the use of multivalued functors to represent 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 [1]). As a consequence a type t is a subtype of another type u when all the values of t are also values of u. Figure 15 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. The same question for birds in (b) is answered positively because there are warm blooded birds.

```
SHOW                    SHOW
warm_blooded('duck')    warm_blooded(bird)
yes                     yes
         (a)                     (b)
```

```
module stars(number:i:t:t1:t2:t3:h:h1:h2:
  h3:m:m1:m2:m3:s:s1:s2:s3:d:d1:iday).
table cometdata(i,h,m,s)= [[0,1,21,2],
  [1,0,58,8],[2,0,34,1],[3,0,7,2]].
table intervaldays(d)=[[0],[1],[2],[3],[4]].
clause interval(i,h1,m1,s1,h2,m2,s2)
  when cometdata(i,h1,m1,s1)
  and cometdata(i+1,h2,m2,s2).
functor time(h,m,s,t) when t=s+60*m+3600*h.
clause hms(t,h,m,s) when s=mod(floor(t),60)
  and m=mod(floor(t/60),60)
  and h=mod(floor(t/3600),24).
clause calcra(h1,m1,s1,h2,m2,s2,d,d1,h,m,s)
  when t1=time(h1,m1,s1)
  and t2=time(h2,m2,s2)
  and t3=t1-((t1-t2)*d1)/d
  and hms(t3,h,m,s).
clause ra(d,h,m,s) when
  d=i*5+iday and interval(*)
  and calcra(h1,m1,s1,h2,m2,s2,5,iday,h,m,s)
  and intervaldays(iday).
```

Figure 14: Interpolation of RA values.

```
MODULE types(STRING:x:herbivore:
  carnivore:bird:fish:invertebrate).
VECTOR herbivore= [elephant,kangaroo].
VECTOR carnivore= [dog,bat].
VECTOR bird= [duck,eagle,ostrich].
VECTOR fish= [shark,dogfish,plaice].
VECTOR invertebrate= [hydra,sponge].
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.
MODULE animals(USING:types, STRING:x).
CLAUSE warm_blooded(x) WHEN x=mammal.
CLAUSE warm_blooded(x) WHEN x=bird.
CLAUSE cold_blooded(x) WHEN x=fish.
```

Figure 15: Functional objects in a type hierarchy.

Since functional domain objects are allowed at the same locations as normal constants the question whether there are animals that are both a mammal and a bird can be stated as in (a) below. The same question for mammals and carnivores is represented as in (b):

```
SHOW mammal=bird  SHOW mammal=carnivore
no                yes
       (a)                  (b)
```

References

[1] 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.

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

[3] 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.

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

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

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

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

[8] 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

[9] 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

1990

Logic, Semantics and Philosophy of Language	
LP-90-01 Jaap van der Does	A Generalized Quantifier Logic for Naked Infinitives
LP-90-02 Jeroen Groenendijk, Martin Stokhof	Dynamic Montague Grammar
LP-90-03 Renate Bartsch	Concept Formation and Concept Composition
LP-90-04 Aarne Ranta	Intuitionistic Categorial Grammar
LP-90-05 Patrick Blackburn	Nominal Tense Logic
LP-90-06 Gennaro Chierchia	The Variablity of Impersonal Subjects
LP-90-07 Gennaro Chierchia	Anaphora and Dynamic Logic
LP-90-08 Herman Hendriks	Flexible Montague Grammar
LP-90-09 Paul Dekker	The Scope of Negation in Discourse, towards a flexible dynamic Montague grammar
LP-90-10 Theo M.V. Janssen	Models for Discourse Markers
Mathematical Logic and Foundations	
ML-90-01 Harold Schellinx	Isomorphisms and Non-Isomorphisms of Graph Models
ML-90-02 Jaap van Oosten	A Semantical Proof of De Jongh's Theorem
ML-90-03 Yde Venema	Relational Games
ML-90-04 Maarten de Rijke	Unary Interpretability Logic
ML-90-05 Domenico Zambella	Sequences with Simple Initial Segments
Computation and Complexity Theory	
CT-90-01 John Tromp, Peter van Emde Boas	Associative Storage Modification Machines
CT-90-02 Sieger van Denneheuvel, Gerard R. Renardel de Lavalette	A Normal Form for PCSJ Expressions
CT-90-03 Ricard Gavaldà, Leen Torenvliet, Osamu Watanabe, José L. Balcázar	Generalized Kolmogorov Complexity in Relativized Separations
CT-90-04 Harry Buhrman, Leen Torenvliet	Bounded Reductions
Other Prepublications	
X-90-01 A.S. Troelstra	Remarks on Intuitionism and the Philosophy of Mathematics, Revised Version
X-90-02 Maarten de Rijke	Some Chapters on Interpretability Logic
X-90-03 L.D. Beklemishev	On the Complexity of Arithmetical Interpretations of Modal Formulae
X-90-04	Annual Report 1989
X-90-05 Valentin Shehtman	Derived Sets in Euclidean Spaces and Modal Logic
X-90-06 Valentin Goranko, Solomon Passy	Using the Universal Modality: Gains and Questions
X-90-07 V.Yu. Shavrukov	The Lindenbaum Fixed Point Algebra is Undecidable
X-90-08 L.D. Beklemishev	Provability Logics for Natural Turing Progressions of Arithmetical Theories
X-90-09 V.Yu. Shavrukov	On Rosser's Provability Predicate
X-90-10 Sieger van Denneheuvel, Peter van Emde Boas	An Overview of the Rule Language RL/1
X-90-11 Alessandra Carbone	Provable Fixed points in $I\Delta_0+\Omega_1$, revised version