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
1986-01 86-02 Peter van Emde Boas
1986-03 Johan van Benthem
1986-04 Reinhard Muskens
1986-05 Kenneth A. Bowen, Dick de Jongh
1986-06 Johan van Benthem

1987 87-01 Renate Bartsch, Martin Stokhof
87-02 Jan Willem Klopp, Roel van der Vlij
87-03 Johan van Benthem
87-04 Víctor Sánchez Valencia
87-05 Eleonore Oversteegen
87-06 Johan van Benthem
87-07 Martijn Harkema, Martin Stokhof
87-08 Renate Bartsch
87-09 Herman Hendriks

1988
1988-01 Michiel van Langenhove Logic, Semantics and Philosophy of Language: Algorithmic Information Theory
1988-02 Yde Venema Expressiveness and Completeness of an Interval Tense Logic
1988-03 Reinhard Muskens Geometric Partiality in Montague Grammar
1988-04 Johan van Benthem Logical Constants across Varying Types
1988-05 Michiel H.M. Smid, Mark H. Overmars General Lower Bounds for the Partitioning of Range Trees
1988-06 Theor M.V. Janssen Type Change in Semantics: The Scope of Quantification and Coordination
1988-07 Theo M.V. Janssen A Mathematical model for the CAT framework of Eurotra
1988-08 Alkene Kleppe A Blisymbolics Translation Program
1988-09 Mathematical Logic and Foundations: Lifschitz’ Realizability
1988-10 Mathematical Logic and Foundations: The Arithemtical Fragment of Martin Lof’s Type Theories with weak $\Sigma$-elimination
1988-11 Dijkstra, Franton Veltman Provability Logics for Relative Interpretability
1988-12 Troelstra On the Early History of Intuitionistic Logic
1988-13 Troelstra Remarks on Intuitionism and the Philosophy of Mathematics
1988-14 L.S. van der Steen, Johan van Benthem Computation and Complexity Theory: Two Decades of Applied Kolmogorov Complexity
1988-16 Michiel H.M. Smid, Mark H. Overmars Maintaining Multiple Representations of Dynamic Data Structures
1988-17 Michiel H.M. Smid, Mark H. Overmars Computations in Fragments of Intuitionistic Propositional Logic
1988-18 Theo M.V. Janssen Machine Models and Simulations (revised version)
1988-19 Peter van Emde Boas, Lex Hendriks A Data Structure for the Union-find Problem having Good Single-Operation Complexity
1988-20 Johan van Benthem, Mark H. Overmars Time, Logic and Computation
1988-21 Michiel H.M. Smid, Mark H. Overmars Multiple Representations of Dynamic Data Structures
1988-22 Leen Torenvliet, Peter van Emde Boas Towards Implementing RL
1988-24 Michiel H.M. Smid, Lex Hendriks Dynamic Predicate Logic, towards a compositional, non-representational semantics of discourse
1988-25 Johan van Benthem, Mark H. Overmars Two-dimensional Modal Logics for Relation Algebras and Temporal Logic of Intervals
1988-26 Johan van Benthem, Michiel H.M. Smid, Mark H. Overmars Language in Action
1988-27 Johan van Benthem, Mark H. Overmars Modal Logic as a Theory of Information
1988-28 Reinhard Muskens Intensional Lambek Calculi: Theory and Application
1988-29 Reinhard Muskens The Adequacy Problem for Sequential Propositional Logic
1988-30 Reinhard Muskens Peirce’s Propositional Logic: From Algebra to Graphs
1988-31 Reinhard Muskens Dependency of Belief in Distributed Systems
1988-33 Johan van Benthem, Lex Hendriks Extending the Lambda Calculus with Subjective Pairing is conservative
1988-34 Johan van Benthem, Michiel H.M. Smid, Mark H. Overmars Rosser Orderings and Fee Variables
1988-35 Michiel H.M. Smid, Mark H. Overmars On the Proof of Solovay’s Theorem
1988-36 Johan van Benthem, Lex Hendriks $\Sigma$-completeness and Bounded Arithmetic
1988-37 Michiel H.M. Smid, Mark H. Overmars The Axiomatization of Randomness
1988-38 Johan van Benthem, Michiel H.M. Smid, Mark H. Overmars Elementary Inductive Definitions in HI: from Strictly Positive towards Monotone
1988-39 Michiel H.M. Smid, Mark H. Overmars Investigations into Classical Linear Logic
1988-40 Michiel H.M. Smid, Mark H. Overmars Provably Fixed Points in $\Delta_1^1 + \Omega_1$
1988-42 Peter van Emde Boas, Lex Hendriks Machine Models and Simulations
1988-43 Peter van Emde Boas, Lex Hendriks On Space Efficient Simulations
1988-44 Leen Torenvliet, Peter van Emde Boas A Comparison of Reductions on Nondeterministic Space
1988-45 Peter van Emde Boas, Lex Hendriks A Parallel Functional Implementation of Range Queries
1988-46 Peter van Emde Boas Finding Isomorphisms between Finite Fields
1988-48 Peter van Emde Boas Honest Reductions, Completeness and Nondeterministic Complexity Classes
1988-49 Peter van Emde Boas On Adaptive Resource Bounded Computations
1988-50 Peter van Emde Boas The Rule Language R/L
1988-51 Peter van Emde Boas Towards Functional Classification of Recursive Query Processing

1989
1989-01 Marianne Kalsbæk
1989-02 G. Wagenaars
1989-03 A.S. Troelstra
1989-04 Jeroen Groenendijk, Martin Stokhof
1989-05 Maarten de Rijke
1989-06 Peter van Emde Boas

1990
1990-01 Marianne Kalsbæk
1990-02 G. Wagenaars
1990-03 A.S. Troelstra
1990-04 Jeroen Groenendijk, Martin Stokhof
1990-05 Maarten de Rijke
1990-06 Peter van Emde Boas

SEE INSIDE BACK COVER
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 transparency 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
<table>
<thead>
<tr>
<th></th>
<th>logical-obj</th>
<th>relational-obj</th>
<th>functional-obj</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(rule-exp)</td>
<td>(scalar-exp)</td>
<td></td>
</tr>
<tr>
<td>extensional-obj</td>
<td>table</td>
<td>map</td>
<td></td>
</tr>
<tr>
<td>(EDB)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>intensional-obj</td>
<td>clause</td>
<td>functor</td>
<td></td>
</tr>
<tr>
<td>(IDB)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Positional</th>
<th>Attributive</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) t(*)</td>
<td>t([a,b,c])</td>
<td>a b c 1 2 3</td>
</tr>
<tr>
<td>(b) t(x,y,#)</td>
<td>t([a=x,b=y])</td>
<td>x y 1 2</td>
</tr>
<tr>
<td>(c) t(x+1,y,#)</td>
<td>t([a=x+1,b=y])</td>
<td>x y 0 2</td>
</tr>
<tr>
<td>(d) t(3,y,#)</td>
<td>t([a=3,b=y])</td>
<td>y void</td>
</tr>
</tbody>
</table>

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
dcl ::= 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 ]

5
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 \hspace{1cm} PRINT 2*v2
\hspace{1cm} 2 \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} 4 \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} 6
\hspace{1cm} (a) \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (b)
PRINT v2*v2 \hspace{1cm} PRINT pow(v2,2)
\hspace{1cm} 4 \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} 4
\hspace{1cm} 6 \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} 9
\hspace{1cm} 9
\hspace{1cm} (c) \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (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 Claustral objects

Claustral objects are defined with one or more clause rules:

\begin{verbatim}
clause-rule ::= CLAUSE clause [ ( attr-list ) ]
             | WHEN rule-exp
             | CLAUSE clause ( * ) WHEN rule-exp
             | CLAUSE clause ( / ) WHEN rule-exp
clause-inv ::= clause [ ( arguments ) ]
\end{verbatim}

In the first defining form the attribute may be omitted when empty; consequently the constants TRUE and FALSE can be defined as claustral objects. In the second (short) defining form, the attributes of the new claustral 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 claustral 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 claustral 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.

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]].

<table>
<thead>
<tr>
<th>PRINT a1(2)</th>
<th>PRINT a2(i)+a3(1,i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>110</td>
</tr>
<tr>
<td>yellow</td>
<td>220</td>
</tr>
<tr>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>PRINT a2(i)+a3(i,1)</td>
<td>PRINT a2(i)+a3(i,j)</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>320</td>
<td>210</td>
</tr>
<tr>
<td>(c)</td>
<td>(d)</td>
</tr>
</tbody>
</table>

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.
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 'sides' 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

8
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 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:

\[
\begin{align*}
\text{SHOW data}(a,b,c) \text{ AND } x=\text{quad}(a,b,c) \\
    x & \quad a & \quad b & \quad c \\
        0 & \quad 1 & \quad 4 & \quad 0 \\
        -1 & \quad 1 & \quad 8 & \quad 7 \\
        -4 & \quad 1 & \quad 4 & \quad 0 \\
        -7 & \quad 1 & \quad 8 & \quad 7
\end{align*}
\]

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 \( x_1,y_1,x_2,y_2 \), the pair \((x_1,y_1)\) denoting the origin and \((x_2,y_2)\) denoting the corner. A user might want a list of these rectangles in an alternative representation system with center points \((x_c,y_c)\) and size pairs \((w_c,h_c)\) 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 \((x_1,y_1), (x_2,y_2)\) to center representation \((x_c,y_c), (w_c,h_c)\) requires solving the constraints \(x_1=x_c-wc, y_1=y_c-hc\), \(x_2=x_c+wc\) and \(y_2=y_c+hc\) for the variables \(x_c, y_c, wc\) and \(hc\). The constraint solver takes care of the solving process:

\[
\begin{align*}
\text{SHOW rectdata}(\#,x_1,y_1,x_2,y_2) \\
\text{AND conv}(x_1,y_1,x_2,y_2,x_c,y_c,w_c,h_c) \\
    x_1 & \quad y_1 & \quad x_2 & \quad y_2 & \quad x_c & \quad y_c & \quad w_c & \quad h_c \\
        0 & \quad 0 & \quad 4 & \quad 4 & \quad 2 & \quad 2 & \quad 2 & \quad 2 \\
        2 & \quad 2 & \quad 7 & \quad 7 & \quad 4.5 & \quad 4.5 & \quad 2.5 & \quad 2.5 \\
        6 & \quad 3 & \quad 9 & \quad 6 & \quad 7.5 & \quad 4.5 & \quad 1.5 & \quad 1.5
\end{align*}
\]

Another user might want to know for a center 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:

\[
\text{SHOW}\ \text{conv}(x_1, y_1, x_2, y_2, 7, 5, 4, 5, 1.5, 1.5) \\
\frac{x_1}{6} \quad \frac{y_1}{3} \quad \frac{x_2}{9} \quad \frac{y_2}{6}
\]

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 ascension (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 ascensions 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:

\[
\text{list ra} \\
\begin{array}{cccc}
0 & 1 & 21 & 2 \\
1 & 1 & 16 & 27 \\
2 & 1 & 11 & 52 \\
\vdots & \vdots & \vdots & \vdots \\
14 & 0 & 12 & 25 \\
\end{array}
\]

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.

\[
\text{SHOW} \quad \text{warm.blooded('duck')} \quad \text{SHOW} \quad \text{warm.blooded(bird)} \\
\text{yes} \quad \text{yes} \\
\]

(a) \quad \text{(b)}
module stars(number:i:t1:t2:t3:h:h1:h2:
h3:m:m1:m2:m3:s:s1:s2:d:d1:iday).
  table comedata(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 comedata(i,h1,m1,s1)
    and comedata(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=15+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:
  VECTORS herbivore= [elephant,kangaroo].
  VECTORS carnivore= [dog,bat].
  VECTORS bird= [duck,eagle,ostrich].
  VECTORS fish= [shark,dogfish,plaice].
  VECTORS 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):

\[
\begin{array}{ll}
\text{SHOW mammal=bird} & \text{SHOW mammal=carnivore} \\
\text{no} & \text{yes}
\end{array}
\]

(a) \hspace{1cm} (b)

References


The ITLI Prepublication Series

1990

Logic, Semantics and Philosophy of Language
LP-90-01 Jaap van der Does
LP-90-02 Jeroen Groenendijk, Martin Stokhof
LP-90-03 Renate Bartsch
LP-90-04 Aarne Ranta
LP-90-05 Patrick Blackburn
LP-90-06 Gennaro Chierchia
LP-90-07 Gennaro Chierchia
LP-90-08 Herman Hendriks
LP-90-09 Paul Dekker
LP-90-10 Theo M.V. Janssen

Mathematical Logic and Foundations
ML-90-01 Harold Schellinx
ML-90-02 Jaap van Oosten
ML-90-03 Yde Venema
ML-90-04 Maarten de Rijke
ML-90-05 Domenico Zambella

Computation and Complexity Theory
CT-90-01 John Tromp, Peter van Emde Boas
CT-90-02 Sieger van Denneheuvel
CT-90-03 Gerard R. Renardel de Lagelette
CT-90-04 Harry Buhrman, Leen Torenvliet

Other Prepublications
X-90-01 A.S. Troelstra
X-90-02 Maarten de Rijke
X-90-03 L.D. Beklemishev
X-90-04
X-90-05 Valentim Shehtman
X-90-06 Valentim Goranko, Solomon Passy
X-90-07 V.Yu. Shavrukov
X-90-08 L.D. Beklemishev
X-90-09 V.Yu. Shavrukov
X-90-10 Sieger van Demmeheul
Peter van Emde Boas
X-90-11 Alessandra Carbone

A Generalized Quantifier Logic for Naked Infinitives
Dynamic Montague Grammar
Concept Formation and Concept Composition
Intuitionistic Categorial Grammar
Nominal Tense Logic
The Variability of Impersonal Subjects
Anaphora and Dynamic Logic
Flexible Montague Grammar
The Scope of Negation in Discourse,
towards a flexible dynamic Montague grammar
Models for Discourse Markers
Isomorphisms and Non-Isomorphisms of Graph Models
A Semantical Proof of De Jongh's Theorem
Relational Games
Unary Interpretability Logic
Sequences with Simple Initial Segments
Associative Storage Modification Machines
A Normal Form for PC/10 Expressions
Generalized Kolmogorov Complexity
in Relativized Separations
Bounded Reductions
Remarks on Intuitionism and the Philosophy of Mathematics,
Revised Version
Some Chapters on Interpretability Logic
On the Complexity of Arithmetical Interpretations of Modal Formulae
Annual Report 1989
Derived Sets in Euclidean Spaces and Modal Logic
Using the Universal Modality: Gains and Questions
The Lindenbaum Fixed Point Algebra is Undecidable
Provability Logics for Natural Turing Progressions of Arithmetical
Theories
On Rosser's Provability Predicate
An Overview of the Rule Language RL/1
Provable Fixed points in $\lambda_\delta + \Omega_1$, revised version