Institute for Language, Logic and Information

TOWARDS IMPLEMENTING RL

Sieger van Denneheuvel
Peter van Emde Boas

ITLI Prepublication Series
For Computation and Complexity Theory CT-88-11

University of Amsterdam

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

TOWARDS IMPLEMENTING RL

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

Received December 1988

Correspondence to:

Faculteit der Wiskunde en Informatica		Faculteit der Wijsbegeerte
(Department of Mathematics and Computer Science)	or	(Department of Philosophy)
Roetersstraat 15		Grimburgwal 10
1018WB Amsterdam		1012GA Amsterdam

Towards implementing RL

Sieger van Denneheuvel & Peter van Emde Boas

Logic and computation theory group
Depts. of Mathematics and computer science, Univ. of Amsterdam.
Nieuwe Achtergracht 166, 1018 WV Amsterdam

Abstract. The integration of numerical constraints and logic programming requires a constraint solver. If constraints and logic programming are furthermore to be integrated with database technology such a constraint solver must be extended so that large amounts of data can be handled efficiently. The integration of these three technologies is the goal of the project reported on in this paper.

1. Introduction

Relational databases, Spreadsheets and Logic Programming are three technologies aimed at processing information in a knowledge based way - users are enabled to express information about the world to be represented in the information processing system in a understandable, structured way. The three technologies have moreover in common that they may be explained using a mathematical model based on relations. Furthermore the three techniques exist in the actual practice of information processing. There exists however no information processing system which integrates them into a joint relational system.

The potential for such an integrated system has been investigated in the context of the Rules Technology project led by Peter Lucas at the IBM San Jose Research Center; see for example [HANS87]. For this purpose the second author had developed in 1985 a language, called **RL** together with a semantic model for its interpretation.

In this paper some issues are described on the way to implementing **RL**. A system RL0 was built to see if implementation of an **RL** system is feasible. An important part of this system is the constraint solver which will be discussed in the subsequent sections. Many aspects of **RL** have not been implemented in the RL0 system. In this paper only those features are described that are present in the current implementation of the RL0 system. In the RL0 system we specifically aim at the integration of algebraic equational constraints and database processing. Although this eliminates a large number of other interesting **RL** features it suffices for exploring new techniques of knowledge processing. For more complete information on **RL** we refer to [ROES87], [VEMD86a] and [VEMD85]. For information on the evaluation of clausal rules on relational databases we refer to [VEMD86b].

2. The RL language

In the **RL** language knowledge can be represented in three different types of rules: *tabular rules, clauses* and *constraints*. Corresponding to these types of rules there are three areas of technology that support that style of knowledge processing in isolation: *database systems, Logic Programming Systems* and *spreadsheets*. Below we will introduce these rules in RL0 syntax with an example.

A main goal of **RL** is to integrate these three technologies in one system. This integration is achieved by providing a relational semantics for these three types of rules. All types of rules determine relations, but the various types of rules interact in a non-trivial way. For more information see [VEMD86a]. Another goal of **RL** is modularization for managing and organizing a large collection of rules in a structured way.

The **RL** language is a declarative representation of knowledge. This means that the user who wants to express knowledge in **RL** rules should not have to worry about control issues in the representation, but only needs to specify what he believes is true in the represented domain.

The representation of the rules should also be *auditable* in the sense that the written text can be inspected by a non-technician in order to convince himself that the rules in the system indeed represent those in the world outside. With the auditability of **RL** two goals may be achieved.
In the first place the authority problem for expert systems (why should we ever believe the answer of an expert system) is reduced to trusting the underlying inference engine because the domain expert can verify the correctness of the rules. Since the **RL** language is declarative rather than imperative, this verification can be done without specific knowledge about the method of inference. Some expert systems, for example OPS5[BROW86], require the knowledge engineer to know details about the implementation of the inference method (inference engine) in order to verify correctness of rules.
Furthermore auditability results in a domain expert who may understand how a problem domain has been represented in **RL** rules. If the knowledge engineer is the same person as the domain expert (I.E., has both active and passive competence about **RL**) then no tedious communication between them is needed which is a benefit for expert system development.

Another purpose of the **RL** project is efficiency in handling large amounts of data. This is to be achieved by an integration with database technology. In this respect the **RL** project shares its motivation with the work reported on in [VEMD86b]. In OPS5, which is an efficient expert system, the knowledge engineer has to bother about restructuring rules in order that efficient evaluation is possible. For example in [BROW86] explicit guide lines are given to avoid big cross products in rules. In **RL** these same issues (e.g. processing joins effectively) arise but are ideally handled by the database query optimizer and not by the knowledge engineer. Other achievements of current database technology like security and recoverability become available as well when the system is integrated with a database system which implements these features.

In [VASS85],[VASS83] four strategies for database access from expert systems are listed:

(1) Elementary Database access within an expert system
(2) Generalized DBMS within an expert system
(3) Loose Coupling of the expert system with an external DBMS
(4) Tight Coupling of the expert system with an external DBMS

The **RL** expert system seems to fit best in classification (4), since the database requirements are not known in advance and the **RL** system works with an external database.

As an example of knowledge represented in **RL** consider the following description for a parallel circuit of two resistors:

```
/* r_tot= circuit resistance, v_tot= circuit voltage

                 i1
                  -->    --------
             -----------|  r1   |-----------
             |           --------           |
o-----------+                               +----------o
             |           --------           |     -->
             -----------|  r2   |-----------     i_tot
                  -->    --------
                   i2
*/

rlmodule electronics.
/* domain declarations */
real:v_tot,i_tot,r_tot,r1,r2,i1,i2,v,i,r,maxwatt.
/* tabular rules */
experimentdata(v_tot,r1,r2)==[
      [9,1200,400],
      [10,1200,400],
      [11,1200,400],
      [12,1200,400]].

/* clauses */
ohmlaw(v,i,r) when v=i*r.
circuit(v_tot,i_tot,r_tot,i1,12,r1,r2) when
      ohmlaw(v_tot,i1,r1) and v_tot*i1<=maxwatt
      and ohmlaw(v_tot,i2,r2) and v_tot*i2<=maxwatt
      and i_tot=i1+i2
      and ohmlaw(v_tot,i_tot,r_tot)
      and maxwatt = 0.25
endmodule.
```

fig. 1

In the above representation of the circuit both the physical laws of basic electricity theory and the practical condition that the resistors should not be overloaded are expressed in terms of simple algebraic equations.

The RL0 system has the capability to determine in the above example for every tuple in the experimentdata the values of all remaining variables in the program. Also in the context of a typical database access the tabular rule defining `experimentdata` may be replaced by a rule like

```
experimentdata(v_tot,r1,r2)== EXTERNAL/X12.b34
```

which describes the import of some database table known to the system by its file-id X12.b34 into the RL0 program; the system now has to determine the other values for every row in this database table (or any subset we like to submit) accordingly.

In the above example we know from elementary physics that the values given in the table `experimentdata` fully determine all quantities. How should the knowledge system determine that this indeed is the case? Had we known in advance that the values `v-tot`, `r1`, and `r2` are the input data, a simple spreadsheet application expressing our knowledge about the solution of this physical problem would have done the job. In the RL0 system we want the system to be capable to figure out this determinedness. The system can moreover switch from one set of fully determining quantities to another. Replacing the above rule by:

```
experimentdata(i1,r1,r2)== EXTERNAL/X12.b35
```

will not disable the system. The system should be as reversible as the ideal relational database; regardless the input values specified, the system should infer everything which can be inferred from these input data (and nothing it should not infer). This requirement of reversibility has led us to the constraint solving problem described below.

3. Constraint solver problem specification

The basic problem that needs to be handled by a constraint solver might be given as follows (Var(X) denotes the set of variables in X):

Input:

System of algebraic constraints:	Cstr
List of known variables:	K
List of unknown variables:	U
List of wanted variables:	W

$K \cup U = \mathrm{Var}(Cstr),\ K \cap U = \varnothing\ ,\ W \subset U$

Output:

List of solutions:

$\mathrm{Soln} = \{\ x = e \mid x \in U, e \in \mathrm{Expr}\ \}$

List of conditions:

Cond= { c | c ∈ Cexp or c=`true` or c=`false` }

The system of equations and conditions in the output must be equivalent to the original system in the sense that the sets of solutions are equal:

Soln ∪ Cond <==> Cstr

The output must also be as informative as possible (note that returning the input unmodified satisfies the specification as given so far). This is expressed by the following two conditions:

(1) For all $c \in$ Cond : $\mathrm{Var}(c) \subset K$

(2) For all $x \in W$: $x{=}e \in \mathrm{Soln} \ \wedge\ \mathrm{Var}(e) \subset K$

The first condition ensures that the condition part only references known variables. The second condition enforces that all wanted variables appear in the solution and that indeed wanted variables are expressed in known variables only.

In the output Expr denotes expressions that can evaluate to a value and Cexp denotes expressions that can evaluate to a truth value.

If `false` is in the Cond list then some inconsistency was derived. The constraints set can then not be solved. For example if 0=1 was derived this would evaluate to `false`. However if `true` is in Cond then some redundancy in the constraints was derived. This means that there was a redundant constraint in the constraints from Cstr. For example if 0=0 was derived this would evaluate to `true`. The inference process of the solver does not allow the derivation of a trivial redundancy for example by substituting an equation in itself.

4. Classification of constraint solvers

Constraint solvers can be classified according to the type of problem they are supposed to deal with. For example, when $K = \emptyset$ condition (1) and (2) from section 3 turn into the conditions:

(3) For all $c \in$ Cond : $\mathrm{Var}(c)=\emptyset$

(4) For all $x \in W$: $x=e \in$ Soln $\wedge$ $\mathrm{Var}(e)=\emptyset$

Condition (3) states that no variables are present in the condition so that if there is a condition in Cond then it is either true or false. Condition (4) states that for all the wanted variables there is a variable-free solution so it must be a value.

Condition (2) can be weakened by allowing unknown variables in the answer, provided they are not on the wanted list:

(5) For all $x \in W$: $x=e \in$ Soln $\wedge$ $\mathrm{Var}(e) \subset K \cup U \backslash W$

With these conditions constraint solvers can be separated in the following classification:

(I) Restricted constraint solver: (3) and (4) hold ($K=\emptyset$)

(II) Solution constraint solver: (3) and (5) hold ($K=\emptyset$)

(III) Condition constraint solver: (1) and (2) hold

(IV) Condition/solution constraint solver: (1) and (5) hold

Clearly (I) represents the most restricted type of a constraint solver and (IV) the most general. The intermediate types (II) and (III) are mutually incomparable.

5. Integrating constraint solvers and databases.

Solving a set of constraints is essentially a two stage process:

(i) Determine solvability of the constraints
(ii) If found solvable then solve the constraints

In this section we will describe how these two tasks can be performed when the constraints are to be repeatedly solved for a large collection of instances presented in a database table. The table contains the possible combinations of values for variables that are to be processed. These variables will form the set of known variables K . Each row (called tuple in the following) from this data table is one combination and assigns one value to each variable in K.

A restricted constraint solver (the least general in this classification) can be used for solving our basic problem in the following way: Simply take one possible combination of values for variables from K. Add equations x=n to the constraint set for all variables x in K and their associated values n. This reformulated problem can now be solved by a restricted constraint solver. Unfortunately the process of constraint solving must be repeated for all tuples from the data table.

In the above strategy the restricted constraint solver answers the question whether constraints are solvable by yes or no. If (3) and (4) hold and no inconsistency was found the answer is yes and the constraints are solved for the particular known values. This results in a unique solution. In the case of a solution constraint solver also an undetermined solution may result where parameters (only unknowns) are present in the answer.

In the case of a condition constraint solver the data table can be processed more effectively. The solver is called once for all tuples with K set to the variables from the data table. When the variables of K are fixed to values of a particular tuple, the actual computation of a value for an unknown is turned into simple evaluation of an expression (namely the expression given in the solution list for the unknown). In the case of a condition/solution constraint solver these expressions still may contain unknown variables which are not on the wanted list. The solution indicates that the system is undetermined, but it has been solved formally; by the time additional information arrives which specifies the remaining unknown values the solution is completed by straightforward evaluation of expressions.

The condition constraint solver answers the question wether constraints are solvable by a condition instead of a {yes,no} answer. The condition can contain (only) known variables and evaluation of the condition yields the yes or no answer. So the actual checking of solvability for a particular tuple is turned into simple evaluation of an expression (namely the condition).

In the following we will represent the results of a query to a condition constraint solver as follows:

```
SELECT     Vars
FROM       Relation
IF         Condition
THEN       Solution
```

The `select` slot contains the variables to be computed and the `from` slot the possible tables from which the data is taken. The `if` and `then` slot correspond to the condition and solution lists from the solver output respectively. In the following the `select` and `from` slots will sometimes be omitted. Note that the above output format suggests the appropriate format of a database query. This is intentional. The constraint solver may produce as its output a syntactically well-formed query which, if processed on the connected database, produces the solutions searched for. This query consists of a select followed by an extension of the retrieved tuples by a suitable set of calculates producing the values for the quantities which have been solved formally.

6. Solver inference engine

The task of the inference engine of a conditional constraint solver is to produce the Cond and Soln lists from the previous section from the list Cstr of constraints. The inference method is based on substitution. It roughly works as follows. Choose an equation E from the set Cstr and select a variable x occurring in E (the combination of E and x is called an (E,x) pair in the following). Then try to isolate the variable x in the equation E (i.e. write the equation in the form x=e ,with x not occurring in e). If this is successful then eliminate x from the system Cstr and Soln by substituting e for x. Remove E from Cstr and add x=e to the list Soln. So now the number of constraints in Cstr is reduced by one. Repeat this process until no constraints in Cstr remain, in which case Cond is set to { }, or until in the constraints from Cstr no (E,x) pair can be found that can be isolated. In this case Cond is set to the current value of Cstr.
In the above process elimination does not mean that the unknown is necessarily evaluated to a value or expressed in known variables. It is possible that the expression substituted for the unknown still contains unknown variables. Also a variable is *only* considered for isolation if it is not a variable from K so known variables are not substituted in the process. The description given here is a simplified version of the algorithm actually used in the solver.

6.1 Isolation methods

In the isolation step several possible heuristic methods can be applied. For isolation to succeed the number of occurrences of x in an equation E must be reduced to one. Simplifying E is an effective heuristic method. Simplification is also important when a transformation to x=e has succeeded because the expression e must be reduced as far as possible (before it is substituted in the other constraints to eliminate x).
The effectiveness of the solver largely depends on the heuristic methods used for isolation, simplification, and the *conflict* resolution strategy (i.e. choosing the best (E,x) pair when more alternatives are possible).
If several (E,x) pairs are in the conflict set then it may be profitable not to just pick one at random but to search for an optimal (E,x) pair. However selecting this optimal pair from all pairs in the conflict set may be quite costly. So a strategy is needed to reduce the size of the conflict set and to find a suboptimal solution. In the current implementation heuristic strategies are used to choose an (E,x) pair from the conflict set.

6.2 Explanation support

For use in an expert system the inference engine should have some kind of explanation available for the derivation of its results. A simple change to the basic inference procedure can provide this explanation. Initially each original constraint in Cstr is associated with an unique number. As the inference process proceeds the new constraints inferred are also assigned a unique number. With each derived constraint a derivation tree is constructed that is composed of numbers of constraints that were used in the derivation. In a derivation tree the leaves are the original constraints for which no further explanation is recorded.
The *justification* set for a derived constraint is the set of all original constraints (i.e. leaves of the derivation tree) that were used for the derivation. The justification set is a flat structure and may be meaningful for the user because its elements may be found in the source RL text.

The above administration improves on having no explanation at all, but the resulting explanation set is not guarantied to be minimal. Especially in the case where an inconsistency or a redundancy was inferred, the user should have a comprehensible explanation. In the RL0 system it is possible to have these minimal explanations by giving the user the possibility to invoke the solver to obtain a minimal explanation set.

7. Some queries for the circuit example

In this section the simple circuit from the beginning is taken as an experiment. The queries below illustrate how known variables can be used to analyse a constraint system. Suppose voltage is applied to the example circuit then the current through the circuit and the total resistance of the circuit can be calculated:

```
?-show circuit(9,i_tot,r_tot,1200,400).
IF   empty
THEN r_tot=300
     i_tot=0.03
```

With other variables given, an answer can also be computed:

```
?-show circuit(v_tot,0.03,r_tot,1200,400).
IF   empty
THEN r_tot=300
     v_tot=9
```

In the constraints written down for the circuit the law for parallel resistors is not given directly, but the total resistance can nevertheless be computed. Now suppose we apply various voltages to the two resistors as specified in the `experimentdata` table. In the table (see fig.1) the voltage is slowly increased and the resistors stay the same. Surely at some high voltage the circuit will break down. Let's run the experiment:

```
?-show experimentdata(v_tot,r1,r2)
     and circuit(v_tot,i_tot,r_tot,r1,r2).

SELECT v_tot,r1,r2,i_tot,r_tot
FROM experimentdata(v_tot,r1,r2)
IF empty
THEN r_tot= r1*r2/(r1+r2)
     i_tot= v_tot/r1 + v_tot/r2
v_tot,r1,r2,r_tot,i_tot
---------------
[9,1200,400,300,0.03]
[10,1200,400,300,0.033]
```

In the output data table the rows correspond to the rows from the experiment data table. Obviously some rows are missing. In the missing rows the constraints were not satisfiable (i.e. the non-smoking condition was violated) so they do not appear in the answer to the query. The exact breakdown voltage can be calculated symbolically:

```
?-show circuit(v_tot,i_tot,r_tot,r1,r2)
and known(r1) and known(r2) and known(v_tot).
```

This query leads to a correct but huge expression which we omit due to the page limit.

Finally if a general relationship between r_tot, r1 and r2 must be found the solver can be directed to find such a relationship by making just these variables known:

```
?-show circuit(v_tot,i_tot,r_tot,r1,r2)
and known(r_tot) and known(r1) and known(r2).
IF    r1*r_tot+r2*r_tot-r1*r2=0
THEN  empty
```

The solver has inferred the law for parallel resistors. Note that this output does not satisfy the requirements for a condition constraint solver (conditions (1) and (2) from section 3) because none of the wanted variables are expressed in known variables. Nevertheless in such cases we would not want to reject the answer because it is informative for the user. The difference between known variables and unknown variables is subtle but important. Consider the following two queries with wanted variables i_tot and r_tot:

```
?-show circuit(v_tot,i_tot,r_tot,r1,r2).
?-show circuit(v_tot,i_tot,r_tot,r1,r2)
     and known(v_tot) and known(r1) and known(r2).
```

In the first query a symbolic solution will probably not express the wanted variables in the other variables and the constraint system is undetermined (i.e. there are too many variables and too few constraints to solve them). In the second query the inference process of the solver is directed to a solution expressing i_tot and r_tot symbolically in terms of v_tot,r1 and r2 and the constraint set is not undetermined.

8. Recursive constraints.

Although in RL recursion is only permitted for clausal rules and not for constraints, the modularization in the language supports the combination of various rule types in a way which makes it possible to define recursive constraints. In order to investigate the problem of solving such recursive constraints RL0 supports them in a more straightforward manner.

For evaluation of recursive clauses a standard method is *iteration on results* [NAQV84]. In this approach the result of an iteration is computed from the result of the previous iteration. The iteration process is started on the empty relation. When the results of two subsequents iterations are the same, iteration stops and the result of the recursive clause is obtained. In [VEMD86b] and [NAQV84] it was pointed out that the iteration on results approach may be quite inefficient for example because the 'selection before join' heuristic can not always be used in optimization of the recursive query.

In [VEMD86b] an alternative general approach for optimizing recursive relational queries, called *iterative compilation*, is introduced. In this approach the next iteration is calculated from the *definition* of the previous iteration instead of the result of the previous iteration. Each iteration is optimized separately and its result is saved. The iterative compilation method makes it possible to optimize, since for instance a select operator can be pushed further down in the relational algebraic tree. Since our strategy for constraint solving involves syntactical manipulation of the constraints this compiled iteration seems appropriate for the integration of recursive clauses with constraint solving as well.

If the constraint system is given in a recursive clause then a legal request would be to find one solution of the constraint system. Recursion can then be stopped when a solution is found. Another legal request would be to find all solutions to a given query. In this case the system must continue to look for more solutions if a solution was found. Since there may be an infinite number of solutions the depth of the query must be bounded.

We illustrate the method by the following example where the predicate mortgage is recursively defined (example from [LASS87]):

```
/* p= principal amount, i= interest rate, b= balance,
   mp= monthly payment, time=duration loan */
CONSTRAINT SYSTEM:
mortgage(p,time,i,b,mp) when
     time <= 1  and  b+mp=p*(1+i).
mortgage(p,time,i,b,mp) when
     1<time  and  mortgage(p*(1+i)-mp,time-1,i,b,mp).
```

fig 2.

Suppose a salesman wants to know the relation between the monthly payment and the principal amount.

```
?-show(mortgage(time,i,b,p,mp)
     and  time=5 and  i=0.1  and  b=0 , 10).

IF   empty
THEN b=0,
     i=0.1,
     mp=p*0.26379
     time=5
```

This can be done with a solution constraint solver (the unknown p appears in the answer for mp). Now the salesman wants to solve the system not for one particular time value but for time as a known variable. This can be done with a condition constraint solver:

```
?-show(mortgage(time,i,b,p,mp) and known(time)
     and known(p) and i=0.1 and b=0 , 3).

IF time<=1
THEN b=0,
     i=0.1,
     mp=p*1.1,

IF time<=2 and time>1
THEN b=0,
     i=0.1,
     mp=p*0.57619,

IF time<=3 and time>2
THEN b=0,
     i=0.1,
     mp=p*0.40211
```

In the examples the query depth was restricted to a maximum by the extra argument in the `show` command.

9. Extension with rules

It should be possible to extend the standard inference system with rules so that the knowledge about operators is extended. For this purpose two types of rules can be added to the constraint solver to make use of the new operators in the same may as the standard operators: reduction rules and isolation rules. The first type represents the simplifications that can be made for the operator (such as actually evaluating the operator to an element of the domain). The second type of rules is used when a variable is isolated in an equation and relates the new operator to other operators to accomplish the isolation.
The two types of rules have the following syntax:

(i) `reduce` Head `to` Body `when` Condition
(ii) `isolate` Head `to` Body `when` Condition

For the head and the body of the reduction rule the usual restrictions from term rewriting systems are in effect. The conditions of both reduction and isolation rules have the form of a conjunct of goals that are evaluated in a Prolog like fashion. They are used as a test that must succeed for the rule to be applicable. Also the conditions can be used to interface with the underlying Prolog system. This is important because not all operators can be represented naturally and efficiently in the above two types of rules. The head and body of an isolation rule are equations that are equivalent if the condition is met.

Example:

```
reduce max(x,x) to x.
reduce max(x,y) to x when
     number(x) and number(y) and x>=y.
reduce max(x,y) to y when
     number(x) and number(y) and x<y.
isolate a+b=c to a=c-b.
```

The first rule says that the maximum of identical expression simplifies to the expression itself and the second and third rule specify how the operator is evaluated if its arguments are numeric. The three rules could be merged into one rule if an appropriate Prolog predicate `pmax` was defined:

```
    reduce max(x,y) to z when pmax(x,y,z).
```

Operator overloading can be used in this approach because the operand types are tested dynamically when the operator is evaluated (late binding). Also a single rule may be used to represent a reduction for several argument types, so minimizing duplication of rules (for example the first rule of the max operator can be used for both numbers and strings).

10. Conclusion

Integration of constraints and logical rules in conjunction with a database system seems to be possible if a constraint solver is extended with the ability to deal with known variables. With this extension a solver does not only produce solutions but can also generate a condition that represents solvability of the constraint system. Heuristic rules are necessary to keep the solving process efficient.
The current RL0 system is written in PROLOG and runs on a Gould computer. The queries and answers figuring as examples in this paper were adapted from the output of this current version.

References.

BROW86	Brownston, L., Farrel, L., Kant, E. & Martin, N., *Programming Expert Systems in OPS5*, Addison Wesley, (1986)
HANS87	Hansen,M.R., Hansen,B.S., Lucas,P. & van Emde Boas,P., *Integrating Relational Databases and Constraint Languages,* Rep IBM Research, RJ 5594 (56904), (1987).
LASS87	Lassez, C., *Constraint Logic Programming, a new general framework for developing languages more powerful than traditional logic programming* languages, Byte , **12** (9), 171-176, (1987).
NAQV84	Naqvi , S.A. & Henschen, L.J., *On Compiling Queries in Recursive First Order Databases*, J. ACM **31**(1), 47-85, (1984)
ROES87	Roessingh, M.J., *RL losgelaten op aanwijzing 111,* Rep. FVI-UvA 87-18 , Dec. 1987.
VASS85	Vassiliou, Y., Clifford, J. & Jarke, M., *Database access requirements of knowledge based systems*, Query Processing in Database Systems, Kim,W. et al. eds., pp. 156-170, (1985).
VASS83	Vassiliou, Y., Clifford, J. & Jarke, M., *How does an expert system get its data?*, Proc. 9th VLDB Conference, Florence Nov 1983, pp. 70-72, (1983).
VEMD85	van Emde Boas, P., *RL, a Language for Enhanced Rule Bases Database Processing*, Working Document, Rep IBM Research, RJ 4869 (51299), Oct. 1986.
VEMD86a	van Emde Boas, P., *A semantical model for integration and modularization of rules*, Proceedings MFCS 12, Bratislava Aug 1986, Springer Lecture Notes in Computer Science **233**, pp. 78-92, (1986).
VEMD86b	van Emde Boas, H. & van Emde Boas, P., *Storing and Evaluating Horn-Clause Rules in a Relational Database*, IBM J. Res. Develop. **30** (1), 80-92, (1986).

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

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