Interpreter for Datalog

Finn B. Pedersen

Master's Thesis
August 1991

Dept. of Computer Science
Technical University of Denmark

Preface

This work constitutes a Master's Thesis for the M.Sc.E. degree at the Technical University of Denmark.

The work was carried out at the Department of Computer Science during the period February 1st through August 31st, 1991, with Professor Jørgen Fischer Nilsson as supervisor.

Finn Bjarne Pedersen

Contents

Chapter 1

Introduction

In the field of answering recursive queries in a deductive database, consisting of function-free clauses, much attention has been paid to the language DATALOG [Bancilhon 86b, Ceri 90, Kemp 88, Kemp 89, Nilsson 90, JFN 91a, Ullman 89a, Vieille 89]. Datalog is a declarative language contrary to PROLOG [JFN 89, Nilsson 90, PDC PROLOG] which again implies the following properties: The syntactic order of rules and literals do not affect the semantics which implies that the answering algorithm and not the user is responsible for termination.

The origin of Datalog

The term "DATALOG" was invented by a group of researchers from Oregon Graduate Center, MCC, and Stanford University. Early publications on DATALOG are [Bancilhon 86a, Bancilhon 86b].

Both pure DATALOG and pure PROLOG have their origin in Horn clauses [Van Emden 76] which are statements of the form: "if A_1 and A_2 and ... A_m are true, then A_0 is true." The syntax for such a statement is:

$$A_0 :- A_1 \ \& \ \ldots \ \& \ A_m.$$

The symbol :- can generally be read "if". Note that most PROLOG versions use a comma where we use the ampersand. The atoms A_i are of the form $p(t_1, \ldots, t_n)$, where p is a n-ary predicate symbol and t_i is a term. The syntax of the term is what makes the difference between DATALOG and PROLOG. In DATALOG the term is either a variable symbol or a constant symbol while in PROLOG the term may also be a function symbol or a predicate symbol. This difference has influenced the development of the two languages.

One has to notice that the logical operators "and" and "or" are both associative and commutative. In Horn clauses this means that the order of clauses and atoms do not influence on the meaning of the statements. A language with this property is said to be *declarative*.

When PROLOG was developed and extensions, like cut and negation as failure, were considered, the language could not be declarative, i.e. the order of statements and atoms was now a part of the semantics for PROLOG.

Languages like DATALOG, PROLOG and LISP have two things in common: They all started by being "pure" languages, i.e. without any extra facilities, with some interesting characteristics and they all needed to be extended for any use on real-life problems.

The interesting characteristics of pure DATALOG are: 1) the language is declarative and 2) termination properties are not user responsible and do not depend of what the statements specify. Please, note that PROLOG does not have these characteristics.

DATALOG does also have some origin in relational database languages [Bancilhon 86a, Ceri 90, Ullman 89a] which have the same characteristics. Queries to a relational database do not have any problems with termination because relations may not be specified recursively.

This work

The aim of this work is to extend DATALOG without violating the characteristics described abow and in such a way that the meaning of programs is clear from the syntax. We will ensure this by using an infix notation for all operators introduced. We will describe the syntax in the Bachus-Normal Form. The syntax of pure DATALOG is extended in a stepwise manner to destinct each extension. To define the well-formedness of extended DATALOG the syntax is also presented as abstract META-IV domains.

We are interested in an effective query answering system. To obtain this, we present how statically to optimize DATALOG programs before the actual query answering evaluation.

Most of the extensions presented in this work can not be specified in the context of pure DATALOG, because they represent infinitely large relations. To deal with this, we will define what is the safe use of these extensions and with a data-flow analysis to determine if a safe use is possible.

We will define safety of DATALOG programs in the terms of guarantee for termination of the query answering algorithm and a safe use of the extensions. In relation to this we will specify, develope and implement an effective query answering algorithm for extended DATALOG.

Overview

In chapter 2 we define the syntax of pure DATALOG. We then give a stepwise introduction to the extensions considered. We are mainly inspired by the extensions considered by [Ceri 90, Ullman 89a, Kemp 89], such as: negation as failure, arithmetical predicates, and comparative predicates. We specify the well-formedness restrictions.

Our extended DATALOG can be considered as a macro expansion of a smaller language. In chapter 3 we show how to transform and optimize the extended DATALOG into a new minimal language with the same expressive power.

In chapter 4 we define safety of minimal DATALOG programs. Safety comprehends legal use of extra predicates and termination of the evaluation method.

In chapter 5 legal use of extra predicates is determined by analyzing the possible data-flow in the program. Manual domain restriction of numeric arguments is introduced to ensure termination of the evaluation method, in cases with recursive defined arithmetical predicates.

Chapter 6 briefly comments on some of the answering algorithms presented in the literatur by describing their application range and origin. The aim of the chapter is to justify our choice of evaluation method.

Chapter 7 is devoted to the specification and construction of an interpreter. The interpreter is based a top-down resolution strategy called *Linear Resolution with Selection Function for Definite Clauses* also known as SLD-resolution. Our method is based on the work of Laurent Vieille [Vieille 89, Ceri 90]. We extend this strategy further to handle both the extra predicates and the manually domain-restriction of bound arguments that we introduce.

Assumptions

The reader is assumed to have knowledge of construction of computer languages [Aho 86], formal specification in META-IV [Bjørner 89, Jones 86], and PROLOG [JFN 89].

Chapter 2

Extending the Syntax of Datalog

Pure DATALOG is declarative and it has some interesting safety properties (see chapter 4). When we extend pure DATALOG with extra facilities it is important that we do not violate these properties.

Extensions of pure DATALOG are considered by [Ceri 90, Ullman 89a], but they both violate the declarative property by not introducing a satisfactory selection function (see section 7.6). Their "missing link" is the data-flow analysis (see chapter 5) which give the necessary information for the selection function to choose a literal that is ready for selection.

There are two reasons why to consider extra predicates: either it is impossible to express the same in pure DATALOG or the operation can be done more efficiently by the computer. By "impossible to express in pure DATALOG" we mean that one would have to specify an infinitely large database to express the same, which is clearly impossible.

2.1 Introduction to the syntax

The syntax of DATALOG is quite similar to that of pure PROLOG. A DATALOG program consists of a finite set of rules, called the *database*, and a *query*.

Rules are of the form

$$A_0 \coloneq A_1 \; \& \; \ldots \; \& \; A_m.$$

And the query is of the form

$$?\!-\; A_0.$$

2.1.1 Pure Datalog

In the case of pure DATALOG the atoms A_i are of the form $p(t_1,\ldots,t_n)$, where p is a n-ary predicate symbol. The arguments are simple terms, i.e. constants or variables. A_0 is the *head* of the rule and the conjunction A_1 & ... & A_m constitutes the *body*. We use the ampersand (&) instead of comma to separate the atoms in the body. This is to emphasize that the atoms are unordered, in contradiction to PROLOG. Rules with $m = 0$, i.e. without *body*, will be referred to as *facts*. There is only one restriction when writing pure DATALOG rules and that is that all variables in the *head* must occur at least once in the *body*, this ensures that the database is *bottom-up evaluable* [Bancilhon 86b] and has therefore the nice property of being safe. Safety means that a given query to the database results in a finite set of solutions, evaluated within a finite amount of time. When a variable never can adopt an infinite set of solutions it is *range restricted*.

The syntax of identifiers and variables in DATALOG is the same as in PROLOG.

When we are talking about a *predicate p*, we refer to the set of rules with the name *p*. If all rules in this set are facts, then the predicate is a *base* predicate and if there are no facts among the rules the predicate is a *virtual* predicate.

Transitive closure example. One of the classical DATALOG examples is the transitive closure program. One can reach a point if there is a direct or indirect path. The query tells whether there are there cycles in the graph or not, i.e. can you reach yourself?

```
?– reach(X,X).
reach(X,Y) :– reach(X,Z) & reach(Z,Y).
reach(a,b).
reach(b,c).
reach(c,d).
reach(c,a).
```

2.1.2 Examples of Extensions

Extracting information example. Consider the following DATALOG program. In a database we have information on a person's address, income, and marital status. From this database we want to extract addresses of married couples who live at the same address and have a large income, say, over Dkr. 300,000.00 a year. The program may look like this:

```
?– query(X).
query(A) :– address_and_income(A,I) & I > 300000.
address_and_income(A,I1+I2) :– person(N1,A,I1) & person(N2,A,I2) &
married(N1,N2).
```

Where person(name,address,income) and married(name,name) are base predicates.

Another example is:

Travel example. The database contains information on bus-, train-, and subwaytrips. The rules specifies that a trip consists of one or more trips by bus, train, or subway. In this case, the query should tell us when to depart on a monday, if we want to arrive in time. The comparative predicates tell that we want to arrive at some time interval before 8 am and that the trip must not last longer than two hours. Please also note that we will not accept to wait longer than 10 minutes between two part-trips.

```
?– query(Depart).
query(Depart) :– trip(ringsted,dth,Depart,Arrival,monday) &
        Arrival < 8–10/60 & Arrival > 8–20/60 & Arrival - Depart < 2.
trip(Start,End,Depart,Arrival,Day) :–
        bus(Start,End,Depart,Arrival,Day).
trip(Start,End,Depart,Arrival,Day) :–
        train(Start,End,Depart,Arrival,Day).
trip(Start,End,Depart,Arrival,Day) :–
        subway(Start,End,Depart,Arrival,Day).
trip(Start,End,Depart,Arrival,Day) :–
        trip(Start,Inter,Depart,InterArrival,Day) &
        trip(Inter,End,InterDepart,Arrival,Day) &
        InterArrival + 10/60 < InterDepart.
```

Both examples illustrate the use of standard predicates, i.e. arithmetic and comparative predicates, in DATALOG. This chapter will in a stepwise manner extend the pure DATALOG into an advanced logic query language for deductive databases. The concrete syntax is specified in the notion of the Backus-Naur Form (BNF) and the abstract syntax is specified in META-IV. All examples will be written in the concrete syntax, while the abstract syntax is used to specify well-formedness which forms the basis of the static syntax check.

2.2 Concrete Syntax of Datalog

We give a complete BNF description of pure DATALOG. This description is then extended in a stepwise manner to distinct each extension. All lines in the BNF description are numbered according to their order in the description of extended DATALOG. This is why some numbers are missing in the following description of pure DATALOG. The missing numbers are introduced with the extensions. Do also notice that some lines which are already defined may be changed more than once.

2.2.1 Pure Datalog

The pure DATALOG is quite simple and easy to understand, see the comments below. Normally 0-ary predicates i.e. predicates without any arguments are not used in connection with databases, but they are common in logic programs, so they will also be allowed here. The standard predicate fail is an example of a 0-ary predicate.

Backus-Naur Form

1.	*⟨DATALOG ⟩*	::= *⟨QUERY ⟩ ⟨DATABASE ⟩*
2.	*⟨DATABASE ⟩*	::= { *⟨RULE ⟩.* }*
3.	*⟨QUERY ⟩*	::= ?– *⟨HEAD ⟩.*
4.	*⟨RULE ⟩*	::= *⟨HEAD ⟩* | *⟨HEAD ⟩* :– *⟨BODY ⟩*
5.	*⟨HEAD ⟩*	::= *⟨ID ⟩* | *⟨ID ⟩ (⟨TERM ⟩* { , *⟨TERM ⟩* }*)*
6.	*⟨TERM ⟩*	::= *⟨VAR ⟩* | *⟨CON ⟩*
7.	*⟨BODY ⟩*	::= *⟨LITERAL ⟩* { & *⟨LITERAL ⟩* }*
8.	*⟨LITERAL ⟩*	::= *⟨ATOM ⟩*
9.	*⟨ATOM ⟩*	::= *⟨HEAD ⟩*
17.	*⟨VAR ⟩*	::= TOKEN
18.	*⟨CON ⟩*	::= TOKEN
21.	*⟨ID ⟩*	::= TOKEN

Comments: A DATALOG program consists of a database and a query. The database is a set of rules, separated by a full-stop. A rule consists of a head and possibly a body. The body is a set of literals, separated by an ampersand (&). The head consists of an identifier and possibly some terms. The syntax of identifiers and variables in DATALOG is the same as in PROLOG. Identifiers begin with a small letter and variables begin with a capital letter. The rest only concerns the pure version of DATALOG: a literal can only be a positive atom and an atom can only be a predicate which is of the same form as a head. Terms can either be variables or constants. Constants in the pure DATALOG are just distinguishable tokens.

2.2.2 Standard predicate: fail

The standard predicate fail is defined by its absence, i.e. fail is a reserved word that can not be defined by the user. Rules that contain a fail can not produce any results. So fail will normally be used to eliminate a rule without erasing it from the source text. As we will see later on, in section 3.5.12, fail plays an important role in optimizing extended DATALOG programs.

Example

```
?– p(X).
p(4).
p(3) :- fail.
p(2).
p(1).
```

2.2.3 Anonymous Variables

Anonymous variables replace variables that only occur once. The anonymous variable can be used in place of any other variable, i.e. in both the query, rule header, and in subqueries. In DATALOG, as in PROLOG, the anonymous variable is represented by a lone underscore. Since anonymous variables may occur in the header, the rules can no longer be guaranteed to return facts. A consequence of this is that a query may give a free variable as result. The purpose of using the anonymous variable is clarity of the programs, i.e. one do not have to invent strange names for variables that are only going to be used once.

Backus-Naur Form

6. ⟨*TERM*⟩ ::= ⟨*VAR*⟩ | ⟨*CON*⟩ | _

Example

```
?– p(X).
p(Y) :- r(_,Y).
r(a,3).
r(b,2).
r(c,1).
```

2.2.4 Stratified Negation

Negation as failure. A negated predicate is true when the positive predicate can not be proved true. This is often referred to as the *Closed World Assumption* **CWA** [Ceri 90, section 11.2.1] which is not an universally valid logical rule, but just a principle. In the context of DATALOG this principle can not be applied to all programs but only a subset called *stratified* DATALOG (see the definition below).

The use and syntax of not in DATALOG are similar to those in PROLOG. fail can never be proved true, so not(fail) is a tautology. Both anonymous variables and variables that only occur once are allowed as terms in negated atoms. In [PDC PROLOG] one may use anonymous variables but not single occurrences of variables in negated atoms.

Stratification. If a predicate *p* does not depend on itself negated *not(p)*, then it is said to be stratified. For stratified programs there exist exactly one fixed point answer which can be evaluated. Stratification is a sufficient condition, but it is not necessary. However, we will only consider stratified programs in this context. The theory and explanation can be found in [Ceri 90, Chapter 11.2].

Non-stratified program. Try to evaluate the program below in a PROLOG manner. If you begin with the rule p the result to the query is X=2,X=3, and X=4, but if you begin with the rule q the result to the query is only X=2. For non-stratified programs, the fixed point may depend on the order in which we evaluate the rules, this is of course unacceptable and non-stratified programs will not be considered further.

```
?- p(X).
p(X) :- r(X) & not(q(X)).
p(2).
q(X) :- r(X) & not(p(X)).
q(1).
r(4). r(3). r(2). r(1).
```

Backus-Naur Form

8. ⟨*LITERAL* ⟩ ::= ⟨*ATOM* ⟩ | not*(*⟨*ATOM* ⟩*)*

2.2.5 Datatypes

In pure DATALOG constants are just some distinct tokens, possible both strings and numbers, but since it is only possible to test whether two terms are equal or different there was no need for typechecking.

In the case of DATALOG only atomic datatypes are considered, i.e. *strings* and some sort of *numbers*. The domain of the numbers will be discussed when we specify the interpreter (section E.1). Complex objects such as *lists* and *function symbols* will not be considered.

The user is free to specify his own types and make a strong typecheck by specifying 1-ary predicates like:

```
domain(true).
domain(false).
```

or

```
domain(X) :- X > 0 & X < 10.
```

Backus-Naur Form

6.	$\langle TERM \rangle$	::= $\langle EXPR \rangle$
11.	$\langle EXPR \rangle$	::= $\langle VAR \rangle$ | $\langle CON \rangle$ | _
18.	$\langle CON \rangle$	::= STRING | NUMBER

2.2.6 Comparative Predicates

Comparative predicates are also referred to as *built-in* predicates in [Ceri 90]. Comparative predicates are used to restrict the set of solutions, by comparison. In this report we use the infix notation instead of the predicate form. The comparative predicates are different from negation in the way that they are base predicates, and have, because of this, no influence on the stratification (see section 2.2.4) of the program.

The kind of queries that can be asked when negation and comparative predicates are involved will be discussed in chapter 4, but consider the following example.

Ordinary citizen example. X is an ordinary citizen if X and the president are two different persons. The predicate ordinary_citizen can only be used to test if a person is the president and not to tell who is an ordinary citizen! The reason is that we will not accept X to occur un-bound in the comparison, since the un-bound variable really corresponds to an infinitely large set of objects.

```
ordinary_citizen(X) :- president(Y) & X != Y.
```

Backus-Naur Form

9.	$\langle ATOM \rangle$	::= $\langle HEAD \rangle$ | $\langle BUILT\text{-}IN \rangle$
10.	$\langle BUILT\text{-}IN \rangle$	::= $\langle EXPR \rangle$ $\langle OPR_1 \rangle$ $\langle EXPR \rangle$
14.	$\langle OPR_1 \rangle$	::= != | < | <= | > | >= | =

Comments

The special predicate symbols have the following names: Not equal (!=), Less Than (<), Less Than or Equal (<=), Greater Than (>), Greater Than or Equal (>=), and Equal (=). Do also note, that only expressions of the same type can be compared, i.e. it makes no sense to compare a number with a string.

2.2.7 Arithmetical Predicates

These abstract descriptions of expressions is found in [Aho 86, page 176]. This structure ensures a deterministic parsing and that multiplication and division have a *higher precedence* than addition and subtraction.

Fibonacci example. The classical Fibonacci function written in DATALOG. The function maps natural numbers into the corresponding Fibonacci numbers.

```
?– fib(5,Y).
fib(0,1).
fib(1,1).
fib(X,Y1+Y2) :– fib(X-1,Y1) & fib(X-2,Y2).
```

Note that arithmetical expressions are recursively defined through subexpressions and can therefore be arbitrarily large.

Backus-Naur Form

11.	$\langle EXPR\rangle$	$::= \langle EXPR\rangle\ \langle OPR_2\rangle\ \langle EXPR_1\rangle \mid \langle EXPR_1\rangle$
12.	$\langle EXPR_1\rangle$	$::= \langle EXPR_1\rangle\ \langle OPR_3\rangle\ \langle EXPR_2\rangle \mid \langle EXPR_2\rangle$
13.	$\langle EXPR_2\rangle$	$::= \langle VAR\rangle \mid \langle CON\rangle \mid -\ \langle EXPR\rangle \mid (\ \langle EXPR\rangle\ ) \mid \_$
15.	$\langle OPR_2\rangle$	$::= + \mid -$
16.	$\langle OPR_3\rangle$	$::= * \mid /$

2.2.8 The Query

The syntax of the *query* is not extended, so only simple terms are allowed in queries.

Backus-Naur Form

3.	$\langle QUERY\rangle$	$::= \text{?–}\ \langle QHEAD\rangle.$
20.	$\langle QHEAD\rangle$	$::= \langle ID\rangle \mid \langle ID\rangle\ (\ \langle QTERM\rangle\ \{\ ,\ \langle QTERM\rangle\ \}^*)$
21.	$\langle QTERM\rangle$	$::= \langle VAR\rangle \mid \langle CON\rangle \mid \_$

2.2.9 Extended Datalog

Finally all BNF rules are assembled to the extended version of DATALOG.

Backus-Naur Form

1.	$\langle DATALOG \rangle$	$::= \langle QUERY \rangle \ \langle DATABASE \rangle$
2.	$\langle DATABASE \rangle$	$::= \{ \ \langle RULE \rangle . \ \}^*$
3.	$\langle QUERY \rangle$	$::= \text{?–} \ \langle QHEAD \rangle .$
4.	$\langle RULE \rangle$	$::= \langle HEAD \rangle \mid \langle HEAD \rangle \ \text{:–} \ \langle BODY \rangle$
5.	$\langle HEAD \rangle$	$::= \langle ID \rangle \mid \langle ID \rangle \ (\ \langle TERM \rangle \ \{ , \ \langle TERM \rangle \ \}^*)$
6.	$\langle TERM \rangle$	$::= \langle EXPR \rangle$
7.	$\langle BODY \rangle$	$::= \langle LITERAL \rangle \ \{ \ \& \ \langle LITERAL \rangle \ \}^*$
8.	$\langle LITERAL \rangle$	$::= \langle ATOM \rangle \mid \text{not}(\langle ATOM \rangle)$
9.	$\langle ATOM \rangle$	$::= \langle HEAD \rangle \mid \langle BUILT\text{-}IN \rangle$
10.	$\langle BUILT\text{-}IN \rangle$	$::= \langle EXPR \rangle \ \langle OPR_1 \rangle \ \langle EXPR \rangle$
11.	$\langle EXPR \rangle$	$::= \langle EXPR \rangle \ \langle OPR_2 \rangle \ \langle EXPR_1 \rangle \mid \langle EXPR_1 \rangle$
12.	$\langle EXPR_1 \rangle$	$::= \langle EXPR_1 \rangle \ \langle OPR_3 \rangle \ \langle EXPR_2 \rangle \mid \langle EXPR_2 \rangle$
13.	$\langle EXPR_2 \rangle$	$::= \langle VAR \rangle \mid \langle CON \rangle \mid - \ \langle EXPR \rangle \mid (\ \langle EXPR \rangle \ ) \mid \_$
14.	$\langle OPR_1 \rangle$	$::= \text{!=} \mid < \mid <= \mid > \mid >= \mid =$
15.	$\langle OPR_2 \rangle$	$::= + \mid -$
16.	$\langle OPR_3 \rangle$	$::= * \mid /$
17.	$\langle VAR \rangle$	$::= \text{TOKEN}$
18.	$\langle CON \rangle$	$::= \text{STRING} \mid \text{NUMBER}$
19.	$\langle ID \rangle$	$::= \text{TOKEN}$
20.	$\langle QHEAD \rangle$	$::= \langle ID \rangle \mid \langle ID \rangle \ (\ \langle QTERM \rangle \ \{ , \ \langle QTERM \rangle \ \}^*)$
21.	$\langle QTERM \rangle$	$::= \langle VAR \rangle \mid \langle CON \rangle \mid \_$

Extended DATALOG examples can be found in appendix A.

2.3 Comments on Parsing

We will not go into implementation details at this moment, but some comments are needed.

The tool for building the interpreter is PROLOG. We use a parser generator (PG) automatically to construct the parser. The input to the PG is based on the BNF description of extended DATALOG. Try to compare the BNF describtion of extended DATALOG above with the PG input in appendix F on page 151.

As output from the PG we get both the parser and the PROLOG domains corresponding to the syntax. From these PROLOG domains we describe the abstract META-IV domains.

The outputs from the parser generator are used as include files in the program. See appendix G where the domains are on page 157 and the parser starts on page 194.

2.4 Abstract Syntax of Datalog

The abstract syntax of extended DATALOG consists of the abstract domains and the well-formedness specification.

2.4.1 Abstract Domains

The abstract META-IV domain equations are either anonymous trees (=) or named (labeled) trees (::). In PROLOG one can name a domain with a function symbol. The domains from the parser generator are used as basis for the abstract domains. This allows us to implement the abstract specifications of well-formedness as directly as possible in PROLOG.

1.	*DATALOG*	$= QUERY \times DATABASE$
2.	*DATABASE*	$= RULE\text{-}\underline{\mathbf{set}}$
3.	*QUERY*	$:: QHEAD$
4.	*RULE*	$:: HEAD \times BODY$
5.	*HEAD*	$:: ID \times TERMLIST$
6.	*TERMLIST*	$= TERM^{*}$
7.	*TERM*	$= EXPR$
8.	*BODY*	$= LITERAL\text{-}\underline{\mathbf{set}}$
9.	*LITERAL*	$= POS \mid NEG$
10.	*POS*	$:: ATOM$
11.	*NEG*	$:: ATOM$
12.	*ATOM*	$= HEAD \mid BUILT\text{-}IN$
13.	*BUILT-IN*	$= NOT\text{-}EQUAL \mid EQUAL$
		$\mid LESS\text{-}THAN \mid LESS\text{-}EQUAL$
		$\mid GREATER\text{-}THAN \mid GREATER\text{-}EQUAL$
14.	*NOT-EQUAL*	$:: EXPR \times EXPR$
15.	*LESS-THAN*	$:: EXPR \times EXPR$
16.	*LESS-EQUAL*	$:: EXPR \times EXPR$
17.	*GREATER-THAN*	$:: EXPR \times EXPR$
18.	*GREATER-EQUAL*	$:: EXPR \times EXPR$
19.	*EQUAL*	$:: EXPR \times EXPR$
20.	*EXPR*	$= PLUS \mid MINUS \mid MULT \mid DIV$
		$\mid VAR \mid CON \mid SIGN$
		$\mid \underline{\text{ANONYMOUS VARIABLE}}$
21.	*PLUS*	$:: EXPR \times EXPR$
22.	*MINUS*	$:: EXPR \times EXPR$
23.	*MULT*	$:: EXPR \times EXPR$
24.	*DIV*	$:: EXPR \times EXPR$
25.	*SIGN*	$:: EXPR$
26.	*VAR*	$:: \mathsf{TOKEN}$
27.	*CON*	$= \mathsf{N}_0 \mid \mathsf{STRING}$
28.	*ID*	$:: \mathsf{TOKEN}$
29.	*QHEAD*	$:: ID \times QTERMLIST$
30.	*QTERMLIST*	$= QTERM^{*}$
31.	*QTERM*	$= QEXPR$
32.	*QEXPR*	$= VAR \mid CON \mid \underline{\text{ANONYMOUS VARIABLE}}$

2.4.2 Abstract Representation of Program

The example below illustrates the transformation done by the parser. The parser reads the *concrete* ascii-file and transforms it to the *abstract* PROLOG domains, here represented by META-IV domains. The syntax tree is unfolded to ease the overview. The unfolding is done in a depth-first manner.

Datalog specification of the Fibonacci function BNF

```
?– fib(X,72).                                % Query.                %
fib(0,1).                                    % Fact.                 %
fib(1,1).                                    % Fact.                 %
fib(X,Y1+Y2) :– fib(X-1,Y1) & fib(X-2,Y2).   % Recursive relation.   %
```

The same specification in abstract domains

1.	$datalog$	$= (query, database)$
2.	$query$	$= QUERY(qhead)$
3.	$qhead$	$= QHEAD(qid, qtermlist)$
4.	qid	$= QID(\mathsf{TOKEN}(fib))$
5.	$qtermlist$	$= < qterm_1, qterm_2 >$
6.	$qterm_1$	$= VAR(\mathsf{TOKEN}(fib))$
7.	$qterm_2$	$= CON(\mathsf{NUMBER}(72))$
8.	$database$	$= \{ rule_1, rule_2, rule_3 \}$
9.	$rule_1$	$= RULE(head_1, \{ \})$
10.	$head_1$	$= HEAD(id_1, termlist_1)$
11.	id_1	$= ID(\mathsf{TOKEN}(fib))$
12.	$termlist_1$	$= < term_{11}, term_{12} >$
13.	$term_{11}$	$= CON(\mathsf{NUMBER}(0))$
14.	$term_{12}$	$= CON(\mathsf{NUMBER}(1))$
15.	$rule_2$	$= RULE(head_2, \{ \})$
16.	$head_2$	$= HEAD(id_2, termlist_2)$
17.	id_2	$= ID(\mathsf{TOKEN}(fib))$
18.	$termlist_2$	$= < term_{21}, term_{22} >$
19.	$term_{21}$	$= CON(\mathsf{NUMBER}(1))$
20.	$term_{22}$	$= CON(\mathsf{NUMBER}(1))$
21.	$rule_3$	$= RULE(head_3, body_3)$
22.	$head_3$	$= HEAD(id_3, termlist_3)$
23.	id_3	$= ID(\mathsf{TOKEN}(fib))$
24.	$termlist_3$	$= < term_{31}, term_{32} >$
25.	$term_{31}$	$= VAR(\mathsf{TOKEN}(X))$
26.	$term_{32}$	$= PLUS(expr_1, expr_2)$
27.	$expr_1$	$= VAR(\mathsf{TOKEN}(Y1))$
28.	$expr_2$	$= VAR(\mathsf{TOKEN}(Y2))$
29.	$body_3$	$= \{ literal_1, literal_2 \}$
30.	$literal_1$	$= POS(atom_1)$
31.	$atom_1$	$= HEAD(id_{31}, termlist_{31})$
32.	id_{31}	$= ID(\mathsf{TOKEN}(fib))$
33.	$termlist_{31}$	$= < term_{311}, term_{312} >$
34.	$term_{311}$	$= MINUS(expr_3, expr_4)$
35.	$expr_3$	$= VAR(\mathsf{TOKEN}(X))$
36.	$expr_4$	$= CON(\mathsf{NUMBER}(1))$
37.	$term_{312}$	$= VAR(\mathsf{TOKEN}(Y1))$
38.	$literal_2$	$= POS(atom_2)$
39.	$atom_2$	$= HEAD(id_{32}, termlist_{32})$
40.	id_{32}	$= ID(\mathsf{TOKEN}(fib))$
41.	$termlist_{32}$	$= < term_{321}, term_{322} >$
42.	$term_{321}$	$= MINUS(expr_3, expr_4)$
43.	$expr_3$	$= VAR(\mathsf{TOKEN}(X))$
44.	$expr_4$	$= CON(\mathsf{NUMBER}(2))$
45.	$term_{322}$	$= VAR(\mathsf{TOKEN}(Y2))$

2.4.3 Auxiliary domains

The following domains are used in connection with specifying the well-formedness of extended DATALOG.

1.	*STRATUM*	$= ID \underset{m}{\rightarrow} \mathsf{INTG}$
2.	*VARSET*	$= TOKEN$<u>**-set**</u>
3.	*MAP*	$= ID \underset{m}{\rightarrow} ARITY$
4.	*ARITY*	$= \mathsf{N}_0 \mid$ <u>CONFLICT</u>
5.	*TYPES*	$= ID \underset{m}{\rightarrow} ARGLIST$
6.	*ARGLIST*	$= TYPE^*$
7.	*TYPE*	$=$ <u>STRING</u> $\mid$ <u>NUMBER</u>

2.4.4 Well-formedness

As mentioned earlier the database must be stratified, we must check that the database is consistent with respect to arity and types, and the standard predicate fail must not be defined. These are the well-formedness restrictions.

```
1.    is-wf-DATALOG(query,db) ≜
 .1     stratified(db) ∧
 .2     aritycheck(query,db) ∧
 .3     typecheck(query,db) ∧
 .4     failcheck(db)
1.5   type: is-wf-DATALOG : DATALOG → BOOL
```

Annotations to <u>**is-wf-**</u>*DATALOG*:

.1 The database must be stratified.

.2 – .3 Check program integrity, i.e. datatypes and predicate arity.

.4 The standard predicate, fail, must not be defined by the user.

End of annotations

Stratification

For each predicate we compute a number, called the *stratum*. In order to stratify the database, each predicate must have at least the same stratum as the positive atoms and at least a stratum that is one larger than the negative atoms that occur in the corresponding rule bodies. Base and standard predicates have a stratum = 1. The following algorithm is from [Ullman 89a, page 134]. The purpose of demanding programs to be stratified is explained in section 2.2.4 and the abstract specification is found in appendix B.

Aritycheck

The aritycheck is quite simple: All rules with common name must also have the same arity. So we construct a table of symbols and their corresponding arity while we parse the syntax tree. When we are through, we check that no conflicts occurred. The abstract specification is found in appendix B.

Different arity example. In this example the predicate p has different arities. Maybe it is possible to give a intuitive meaning to this kind of programs, by identifying predicates by name and arity, but we will not do that.

```
?– p(X,Y).
p(1). p(1,2). p(1,2,3).
```

Typecheck

As for the pure DATALOG we will allow predicate arguments to be of both types. Comparative predicates will accept expressions of both types, but not at the same time of course. Arithmetical predicates will only accept numeric operands. We have not specified the typecheck. However, we are convinced that it is possible. We estimate the complexity of such an algorithm to be compared with that of the data-flow analysis which is described in section 5.

Example of type conflict

```
?– p(a,1).
p(X,Y) :- X < Y.
```

Fail predicate may not be defined by the user

In this example fail can be proved true, which is absurd and unacceptable. The abstract specification is found in appendix B.

```
?– fail.
fail.
```

2.5 Summary

In this chapter we have extended DATALOG with some well known features from PROLOG. The extension is done in a stepwise manner to distinct each separate extension.

The extensions are:

- negation as failure, not().

- datatypes (string and number),
- arithmetical predicates (+, −, ∗, and /),
- anonymous variables (_),
- comparative predicates (<=, >=, <, >, =, !=), and
- the standard predicate fail.

We have defined the well-formedness of DATALOG programs, which consist of controlling program integrity on four subjects: the program must be stratified, all rules with the same name must also have the same arity, the typecheck, and the predicate fail must not be defined by the user.

In the extended DATALOG the user is free to specify relations without any consideration of effectiveness. This is what makes DATALOG a true declarative language. DATALOG examples can be found in appendix A.

2.5.1 Other extensions

[Ceri 90] consider introducing complex objects, aggregate functions, nondeterminism, uncertainty and null values. Complex objects are also considered by [Bancilhon 89].

We do not consider these extensions for two reasons: either will the extensions violate the interesting properties of the pure DATALOG or the use of the extesion is doubtful.

Chapter 3

Optimizing Extended Datalog Programs

3.1 Introduction

The extended DATALOG can be considered as a macro expansion of a smaller language. We call this language *minimal* DATALOG. By transforming into the smallest possible language with the same expressiveness we obtain three advantages: 1) optimized programs, 2) a larger set of possible queries, and most important of all, 3) the interpreter becomes much simpler.

Some transformations can be done in one pass of the program and others need several passes. First we transform general expressions into simple expressions. This is done in one pass. Then we optimize the program by unfolding non-recursive rules and then eliminating tautologies and absurdities, which is done in several passes until no further changes are possible. At last we rename all variables that occur only once in a rule to anonymous variables, also in one pass.

Minimal DATALOG is actually a subset of extended DATALOG, i.e. it is possible to write extended DATALOG programs which will not be changed by the transformation since these programs are minimal already.

3.2 Rewriting Expressions

The advantage of having arbitrarily large numeric expressions as terms in predicates is a more userfriendly language, but it is difficult to handle these expressions during the evaluation of the program.

We will show how to break larger expressions down into simple expressions, i.e. expressions with only one operator and two simple terms, i.e. constants or variables.

3.2.1 Hypothesis

Our hypothesis is that it is possible, without loss of generality and expressiveness, to have only simple terms as predicate arguments. See the following example for further clarity. As a proof we give the transformation function which transforms extended DATALOG into extended DATALOG without compound expressions.

The number of possible operators is reduced from four to two, because division can be expressed by multiplication and subtraction by addition, i.e. $Y - Z = X$ is equivalent to $X + Z = Y$. Where X, Y, and Z are simple terms. Note that these operations do not show how to evaluate the result, they represent a relation between the variables.

To simplify the program further, before evaluation, the possible comparative predicates are reduced. GREATER THAN and GREATER THAN OR EQUAL are transformed into LESS THAN and LESS THAN OR EQUAL respectively by swapping the variables.

Do also note that there are no changes in the concrete BNF syntax, we just use a subset of it, but the transformation results in a new abstract syntax. This syntax will be referred to as the *minimal* DATALOG. *Minimal* DATALOG has the same expressiveness as *extended* DATALOG, because we only remove standard predicates that can be constructed by other standard predicates. SUBTRACTION by ADDITION, GREATER THAN by LESS THAN, and so on.

3.2.2 Example

Natural logarithm of two. This computes the sum of a finite sequence of constants. The numbers involved in this example are either represented as rational numbers or some floating point numbers. The sum gives an approximation to the natural logarithm of two.

Extended datalog

```
?- ln2(X).
aux_ln2(X,0)   :- X > 10000.
aux_ln2(X,1/X - 1/(X+1) + Y) :- aux_ln2(X+2,Y).
ln2(X)          :- aux_ln2(1,X).
```

The first and the second rule are changed during the transformation. The transformation results in the following program. The syntax is still extended DATALOG but with many small expressions instead of a few large expressions.

Minimal datalog

```
?- ln2(X).
aux_ln2(X,0)   :- 10000 < X.
aux_ln2(X,V2) :- aux_ln2(V1,Y) &
                 V3 + Y = V2 &
                 V3 + V5 = V4 &
                 V4 * X = 1 &
                 V5 * V6 = 1 &
                 X + 1 = V6 &
                 X + 2 = V1.
ln2(X)         :- aux_ln2(1,X).
```

The answer to the query and the automated transformation can be found in the examples' appendix at page 85.

3.2.3 Transformation of numeric expressions.

An expression is a tree structure. It consists of either a constant, a variable or an operator with two subexpressions. The idea is to parse the tree in the following way:

- If the tree is a variable or a constant, then return this variable or constant with an empty set.
- Otherwise the node consists of two subexpressions and an operator *opr*
  1) introduce a *new* variable *V*.
  2) parse the subexpressions recursively. The subexpressions return with the two variables *V1* and *V2* and the set of standard predicates they have constructed.
  3) bind the new variable to the expression by introducing this standard predicate: *V1 opr V2 = V*. This standard predicate is returned together with the standard predicates from the subexpressions and the new variable V.

3.2.4 Transformation function

The specification of the transformation function $\mathcal{T}$-DATABASE is to be found in section C.1. The specification of the expression transformation $\mathcal{T}$-EXPR is found on page 119.

3.3 Introduction to optimization

The extended DATALOG is a powerful language. We have given some examples to show the expressiveness, but what is at least as interesting is that the extended DATALOG has a great potential for optimization. The following example and comments illustrate this potential.

3.3.1 Example

Unfolding. This program is usually not considered as a correct DATALOG program and programs of this type are not discussed in the literature used as basis for this report. See the bibliography at page 82.

The variables X and Y are not bound in the call of the second rule. To evaluate this we have to substitute the head of q and r with their body in all occurrences and make the necessary bindings and matches. This is called *unfolding* and it can be applied to all non-recursive predicates.

```
?- p(X,Y).
p(X,Y)       :- q(X,Z,Z) & r(Z,Z,Y).
q(1,X,Y)     :- X <= Y.
r(X,Y,2)     :- X <= Y.
```

The unfolding results in the following program.

```
?- p(X,Y).
p(X,Y)       :- X = 1 & Z <= Z & Y = 2.
```

This can obviously be optimized further. X and Y are bound to constants by the standard predicate equal and can therefore be substituted by these values. Z <= Z is a tautology and can be removed. The result is an optimized and evaluable program.

```
?- p(X,Y).
p(1,2).
```

This example (not the unfolding) is from [Bancilhon 86b] who concludes: "The query p(X,Y) is safe (the answer is { p(1,2) }), but there is no general evaluation strategy which will compute the answer and terminate".

We have now disproved this remark by introducing unfolding as a general optimization strategy.

3.3.2 Logical Identities

The Logical Identities below are found in [Stanat 77, page 15], where they are used in the context of mathematical reasoning and boolean logic. Each identity can be established by constructing a truth table.

The logical identities are the rules that we use to transform the program without changing the semantics of the program. Between logical identities one normally writes a biimplication sign, but when we just optimize in one direction, we use an implication arrow to show the direction.

- $(P \vee P) \Longrightarrow P$. Idempotence of $\vee$ is used when we transform the concrete syntax into the abstract syntax. We transform lists into sets. The result is that if a rule in the program occurs more than once, the duplicates are removed.
- $(P \wedge P) \Longrightarrow P$. Idempotence of $\wedge$ is used when we transform the concrete syntax into the abstract syntax. We transform lists into sets. The result is that if a literal in a body occurs more than once, the duplicates are removed.
- $(P \vee Q) \Longleftrightarrow (Q \vee P)$. Commutativity of $\vee$ means that we are free to choose the order in which we will evaluate the rules in the program.
- $(P \wedge Q) \Longleftrightarrow (Q \wedge P)$. Commutativity of $\wedge$ means that we are free to choose the order in which we will evaluate the literals in the rule body, as long as we obey the adornments of the predicates. (Will be discussed further in section 4.2)
- $[(P \vee Q) \vee R] \Longleftrightarrow [P \vee (Q \vee R)]$. Associativity of $\vee$ means that the order of evaluation of rules is irrelevant for the result.
- $[(P \wedge Q) \wedge R] \Longleftrightarrow [P \wedge (Q \wedge R)]$. Associativity of $\wedge$ means that the order of evaluation of literals is irrelevant for the result. This is only true when commutativity of $\wedge$ is valid. See above.
- $\neg(P \vee Q) \Longrightarrow (\neg P \wedge \neg Q)$, $\neg(P \wedge Q) \Longrightarrow (\neg P \vee \neg Q)$. DeMorgan's Laws are used in negative unfolding of non-recursive predicates.
- $[P \wedge (Q \vee R)] \Longleftrightarrow [(P \wedge Q) \vee (P \wedge R)]$. Distributivity of $\wedge$ over $\vee$. Used in unfolding.
- $[P \vee (Q \wedge R)] \Longleftrightarrow [(P \vee Q) \wedge (P \vee R)]$. Distributivity of $\vee$ over $\wedge$. Used in unfolding.
- $(P \wedge T) \Longrightarrow P$, where T is a tautology. Is used in optimization.
- $(P \vee F) \Longrightarrow P$, where F is fail. Rules that fail can be removed.

- $(P \wedge F) \Longrightarrow F$, where F is fail. Is used in optimization.
- $\neg(\neg(P)) \Longrightarrow P$. Double negation is used in negative unfolding.

3.4 Optimization of Datalog

The optimization of DATALOG consists of several strategies which depend on each other, i.e. some of the strategies does only optimize the program if some of the other strategies are present. Each strategy is specialized with one purpose, it transforms by eliminating and possibly introducing different constructions. By iterating this process until the program reaches a fixed point, we obtain a specialized program. The specialized program will belong to the syntax of minimal DATALOG. In this section we specify the abstract syntax of minimal DATALOG and we describe the iterative optimization strategy.

We split the *database* into an *extensional* database (EDB), consisting of facts, and an *intensional* database (IDB), without any facts, to obtain a more realistic evaluation function. By *realistic* we mean closer to what we have to write when we construct the interpreter.

3.4.1 Abstract Domains

The minimal syntax differs from the extended syntax in the following rules.

Differences of extended and minimal Datalog

- The database is split into an extensional database and an intensional database.
- Negative literals can no longer be *standard predicates*, but only *heads*, i.e. the negative standard predicates are transformed into positive ones.
- Terms can no longer be expressions, but only simple terms, i.e. either constants, variables, or anonymous variables.
- The EQUAL predicate does only occur in connection with simple expressions, i.e. with only one operator. Bindings between two variables are removed by substitution. This use of EQUAL and an operator forms an arithmetical predicate.
- Fewer operators. GREATER THAN, GREATER THAN OR EQUAL, DIVISION, and SUBTRACTION have already been transformed into other operators.

Abstract Domains

1.	*DATALOG*	=	*QUERY* × *DATABASE*
2.	*DATABASE*	=	*EDB* × *IDB*
3.	*EDB*	=	*ID* $\underset{m}{\rightarrow}$ *TUPLE*-**set**
4.	*TUPLE*	=	*CON* *
5.	*IDB*	=	*RULE*-**set**
6.	*QUERY*	::	*QLITERAL*
7.	*RULE*	::	*HEAD* × *BODY*
8.	*HEAD*	::	*ID* × *TERMLIST*
9.	*TERMLIST*	=	*TERM* *
10.	*BODY*	=	*LITERAL*-**set**
11.	*LITERAL*	=	*POS* ∣ *NEG*
12.	*POS*	::	*ATOM*
13.	*NEG*	::	*HEAD*
14.	*ATOM*	=	*HEAD* ∣ *BUILT-IN*
15.	*BUILT-IN*	=	*LESS-THAN* ∣ *LESS-EQUAL*
			∣ *NOT-EQUAL* ∣ *EQUAL*
16.	*LESS-THAN*	::	*TERM* × *TERM*
17.	*LESS-EQUAL*	::	*TERM* × *TERM*
18.	*NOT-EQUAL*	::	*TERM* × *TERM*
19.	*EQUAL*	::	*EXPR* × *TERM*
20.	*EXPR*	=	*PLUS* ∣ *MULT* ∣ *SIGN*
21.	*PLUS*	::	*TERM* × *TERM*
22.	*MULT*	::	*TERM* × *TERM*
23.	*SIGN*	::	*TERM*
24.	*TERM*	=	*VAR* ∣ *CON* ∣ ANONYMOUS VARIABLE
25.	*VAR*	::	TOKEN
26.	*CON*	=	*R-CON* ∣ *S-CON*
27.	*R-CON*	::	NUMBER
28.	*S-CON*	::	STRING
29.	*ID*	::	TOKEN
30.	*QLITERAL*	=	*QPOS*
31.	*QPOS*	::	*HEAD*

3.4.2 Well-formedness

We do not explicitly specify well-formedness for the transformed and optimized programs. We claim that well-formedness is not violated during the transformation and optimization. This means that the program is still stratified, arity and type integrities are not violated, and we do not introduce the standard predicate fail as a fact.

3.5 Iterative optimization strategy

Each strategy is described by what it transforms, eliminates and/or introduces. The following introduction gives an informal description.

Iterative optimizing strategy.

1. Dependency graph. The order of optimizing the rules is determined by the graph. The Graph is used by the following strategy.
2. Remove rules, that do not concern the query. By removing these rules as early as possible, we avoid optimizing rules with no affect on the result.
3. Split the database. Move all facts to the extensional database. By splitting the database we can concentrate on optimizing virtual rules.
4. Make rule graph. Order all rules according to dependency graph.
5. Unfolding of non-recursive predicates. This is the most difficult and powerful strategy. Unfolding means to substitute the head with body and make the proper bindings. The resulting intensional database will then only consist of recursive predicates and possibly one non-recursive predicate which has the same name as the query.
6. Optimizing numeric operations. We try to evaluate statically as much as possible.
7. Transform negative standard predicates into positive standard predicates.
8. Optimizing comparative predicates. Tautologies and failures can be optimized effectively.
9. All bindings of simple terms with EQUAL can be removed by substituting or comparing them.
10. Converting unknown predicate names to the standard predicate fail.
11. Uninitialized recursive predicates will always fail.
12. Remove fail. If fail occurs as a positive literal then remove the rule. The literal not(fail) is a tautology and can be removed.

3.5.1 Dependency graph

The dependency graph needs only to be reevaluated if rules are removed from the program. If a rule is removed, then there might be other rules that can not affect the query any longer. These will be removed in the next step. Our dependency graph is strongly inspired by the rule/goal graph presented in [Bancilhon 86b].

A predicate is just a name. This is the name occurring in the head of some of the rules in the database.

Definition: Predicate dependency. We define *dependency* recursively by saying that a predicate p depend on another predicate q if there exists a rule, with p in the head and a literal in the body, which is either q or another predicate that depend on q

Definition: Recursive predicate. A predicate is *recursive* if it depends, directly or indirectly, on itself. Two or more predicates are *mutually* recursive if they depend on each other.

Auxiliary domains

The following auxiliary domains are needed to evaluate the dependency graph.

1. $DEPENDGRAPH = DEPEND\text{-}OF^{*}$
2. $DEPEND\text{-}OF = ID \xrightarrow[m]{} ID\text{-}\underline{\mathbf{set}}$

Algorithm

To each predicate we determine what predicates it depends on. Then we devide this set of dependencies into subclasses, such that predicates that mutually depend on each other are in the same class. Then we order the classes such that each class only depend on classes left to itself. We have thereby obtained the dependency graph.

3.5.2 Remove rules, that do not concern the query

This strategy does not transform or introduce anything. It reduces the set of rules to be optimized. First we locate the name of the query in the dependency graph. All elements to the right of this position can be removed immediately and the rest of the graph is pruned in a top-down recursive manner by only selecting elements that will affect the query directly or indirectly. Note that if the query is fail or another unknown predicate then all rules and facts are removed from the database and the answer to the query is NO.

3.5.3 Split the database

This strategy does not transform or introduce anything. It reduces the set of rules to be optimized. *Facts* are the most optimized rules at all and what we try to obtain by the optimization. Facts can be introduced by removing simple bindings. In minimal DATALOG the database consists of an EDB part and an IDB part. Whenever a fact occurs in the IDB, it is moved to the EDB.

3.5.4 Make rule graph

The information in the dependency graph allow us to categorize the predicates occurring in the database.

- 1. category: Predicate names occurring on the right hand side but not on the left hand side of any rules, i.e. the predicate has no specification. However this should never occur if the database is optimized but we will consider this case anyway, because it enables us to test the interpreter with programs that are both optimized and not.
- 2. category: Consists of non-recursive rules, and
- 3. category: Consists of recursive and mutually recursive rules.

Predicates with rules in the 3. category do normally also have rules in the 2. category to initialize the recursion. In the rule-graph the rules are ordered and categorized according to the dependency graph.

To optimize the program in an effective manner, we have to do it in the order given by this graph.

Auxiliary domains

The following auxiliary domains are needed to evaluate the rule graph.

1.	*RULEGRAPH*	= *GROUP**
2.	*GROUP*	= *INIT* | *STEP* | *REPEAT*
3.	*INIT*	:: *ID*
4.	*STEP*	:: *IDB*
5.	*REPEAT*	:: *IDB*

Our rule graph corresponds to the *reduced rule/goal graph* presented in [Bancilhon 86b] where more theory can be found.

3.5.5 Unfolding of non-recursive predicates

Unfolding means to substitute the head with the body and make the necessary bindings. The resulting program will then only consists of recursive predicates and possibly one non-recursive predicate which has the same name as the query.

Introduction to unfolding

We first consider the case of boolean logic which is quite similar to 0-ary predicates in DATALOG. Unfolding in this case means substituting the head with the body and no matching of variables is needed.

Unfolding is split into two cases;

- Positively. When the non-recursive predicate occurs as a positive literal in the body of some rule, and
- Negatively. When the predicate occurs negated, i.e. in `not`.

Consider this logical assertion and the positive unfolding of e

$$\begin{array}{l}
e \land f \\
e = (a \land \neg b) \lor (c \land \neg d) \\
\Downarrow \\
((a \land \neg b) \lor (c \land \neg d)) \land f \\
\Downarrow \\
(a \land \neg b \land f) \lor (c \land \neg d \land f)
\end{array}$$

Now consider the negative unfolding of e

$$\begin{array}{l}
\neg e \land f \\
e = (a \land \neg b) \lor (c \land \neg d) \\
\Downarrow \\
\neg((a \land \neg b) \lor (c \land \neg d)) \land f \\
\Downarrow \\
\neg(a \land \neg b) \land \neg(c \land \neg d) \land f \\
\Downarrow \\
(\neg a \lor b) \land (\neg c \lor d) \land f \\
\Downarrow \\
(\neg a \land \neg c \land f) \lor (\neg a \land d \land f) \lor (b \land \neg c \land f) \lor (b \land d \land f)
\end{array}$$

In each step we have used the logical identities we described in section 3.3.2.

General application

The question is whether this unfolding is valid or not in the general case, i.e. do we get the same answers when we unfold.

We claim that if the query does not violate the range restriction demand (section 4.2) then the result will be the same but some queries which are not range restricted are made range restricted by negative unfolding. We accept this as an optimization result. The following example illustrates this special case. The predicates good and sweet are base predicates.

Not range restricted query

```
?– nice(X).
nice(X)        :– not(bad(X)).
bad(X)         :– not(good(X)).
bad(X)         :– not(sweet(X)).
```

This program is not range restricted because the variable X in the first rule is demanded to be instantiated by the literal not(bad(X)) but the variable is not bound to any value by the query nice(X). After the unfolding we obtain the following program.

Range restricted query

```
?– nice(X).
nice(X)        :– good(X) & sweet(X).
```

This program is range restricted because no variables are demanded to be instantiated at any time. The non-recursive predicate bad is removed during the unfolding.

Comment

Most logicians would probably state that the semantics of the original program and the transformed/unfolded program are different. We state that it depends on how one interpret non-recursive predicates. There are two possibilities:

- Non-recursive predicates are **logical statements** which implies that negative unfolding may change the semantics of the program.
- Non-recursive predicates are **macroes** with the purpose of improving the overview of the program which implies that the unfolding is a part of the semantics.

Note that the expressive power of DATALOG is independent of whether non-recursive predicates are allowed or not. We prefer the second interpretation of non-recursive predicates.

3.5.6 Optimizing numeric operations

As an introduction to optimizing numeric operations, we give several examples that need no comments. ⇒ means optimized into. X, Y, and Z are meta variables and a, b, and c are arbitrary numbers if nothing else is mentioned.

Simple optimizations.

X + b = a ⇒ X = c, where c=a–b.
X + X = X ⇒ X = 0.
X + a = X ⇒ fail, if a≠0.
Y + 0 ⇒ X = Y.
X + X = a ⇒ X = b, where b=a/2.
b + c = a ⇒ fail If the assertion is false.
b + c = a ⇒ If this assertion is true we remove it.
Y ∗ 1 = X ⇒ X = Y.
X ∗ 0 = 0 ⇒ true for all X.
Y ∗ 0 = X ⇒ X = 0, for all Y.
X ∗ X = X ⇒ X = 1 or X = 0.
X ∗ X = 0 ⇒ X = 0.
a<0, X ∗ X = a ⇒ fail.
a>0, X ∗ X = a ⇒ X = -$\sqrt{a}$ or X = $\sqrt{a}$.
X = –X ⇒ X = 0.

Compound optimizations.

Y + a = X & Z + b = Y ⇒ Z + c = X, where c=a+b, if and only if Y only occur in the two shown literals.
-Y = X & -Z = Y ⇒ X = Z, if and only if Y only occur in the two shown literals.

More research is need in the area of compound optimizations.

Duplication of operations caused by commutative operators.

Y + Z = X & Z + Y = X ⇒ Y + Z = X.
Y ∗ Z = X & Z ∗ Y = X ⇒ Y ∗ Z = X.

3.5.7 Transform negative comparative predicates

All negated comparative predicates can be transformed into positive comparative predicates with the following meaning.

```
not(X = Y) ⇒ X != Y
not(X < Y) ⇒ Y <= X
not(X <= Y) ⇒ Y < X
not(X != Y) ⇒ X = Y
```

Where X and Y are simple terms, i.e. variables or constants. See specification in appendix C.2.1.

Example

```
?- p(3).
p(X) :- not(X = 4 + Y) & q(Y).
q(1).
```

Is transformed into:

```
?- p(3).
p(X) :- X != V1 & V1 = 4 + Y & q(Y).
q(1).
```

3.5.8 Optimizing comparative predicates

Tautologies are eliminated, while absurdities introduce the standard predicate fail. Bindings may be hidden in pairs of comparative predicates. By introducing the binding it can be further optimized. Also, redundant comparative predicates can be removed.

- Tautologies: X <= X, X = X.
- Absurdities: X < X, X != X, X < Y & Y < X, X < Y & Y <= X.
- Hidden binding: X <= Y & Y <= X ⇒ X = Y.
- Redundance: X <= Y & X < Y ⇒ X <= Y.
- Redundance: X != Y & X < Y ⇒ X < Y.
- Redundance: X != Y & X <= Y ⇒ X < Y.

Example

```
?– p(X).
p(X) :– X < X & q(X).
q(1).
```

Is transformed into:

```
?– p(X).
p(_) :– fail.
```

3.5.9 Remove simple bindings

Bindings can be eliminated by substitution. X and Y are meta variables and a and b are arbitrary constants.

- $X = Y$: All occurrences of X are substituted by Y.
- $X = a$ or $a = X$: All occurrences of X are substituted by a.
- $a = b$: If a is identical with b then it is a tautology and can be removed, otherwise a fail is introduced.

See specification in appendix C.2.2.

Example

```
?– p(X).
p(X) :– X = a.
```

Is transformed into:

```
?– p(X).
p(a).
```

3.5.10 Transform unknown predicate names to fail

If a query or a subquery can not be proved because there exist no rules of that name, then replace that query/subquery with the standard predicate fail.

Example

```
?– p(X).
p(X) :– q(X).
```

Is transformed into:

```
?– p(X).
p(_) :- fail.
```

3.5.11 Uninitialized recursive predicates

If all rules of a set of mutual recursive predicates are directly or indirectly recursive, i.e. no rules initialize the recursion, then add a fail to all these rules. Consider the following DATALOG program with just two rules.

```
?– p(X).
p(X) :– q(X).
q(X) :– p(X).
```

If a recursive definition is not initialized with some values, it can never produce any solutions.

3.5.12 Remove fail

fail can either occur as a positive or a negative literal. If it is positive then remove the rule from the database, because it can never produce any solutions. If it is negative, i.e. not(fail), it is a tautology and can be removed from the rule body.

Example

```
?– p(X).
p(X) :– not(fail) & q(X).
q(1).
```

Is transformed into:

```
?– p(X).
p(X) :– q(X).
q(1).
```

3.6 Introduce anonymous variables

If a variable name is only used once in a rule, then we rename it to an anonymous variable. We do this for three reasons: 1) when we find a variable that only occurs once, we can warn the user of the existence of such a variable, 2) to ease the computation of possible adornments and 3) to optimize a top-down evaluation strategy, i.e. solutions to an anonymous variable need only to be either *failure* or *success*.

See the specification in appendix C.3.

Example

```
?– p(X).
p(X) :– q(X,Y).
q(1,2).
```

Is transformed into:

```
?– p(X).
p(X) :– q(X,_).
q(1,2).
```

The variable Y is substituted with an anonymous variable.

3.7 Summary

In this chapter we show how to optimize DATALOG programs. We try statically to evaluate as much as possible. The aim of this is to help the query answering algorithm to focus on the relevant answers and thereby minimize the irrelevant work. After the optimization, the program now belongs to the syntax of the minimal DATALOG. Since *minimal* DATALOG is a subset of *extended* DATALOG, but with the same expressiveness, we obtain more homogeneous programs. It is obvious that minimal DATALOG programs are more easy to evaluate than extended DATALOG programs in general.

3.7.1 Partial Evaluation

A widely accepted optimization strategy is *partial evaluation* [Bondorf 90, Fuller 88]. Partial evaluation means to specialize a program according to the known input data. In DATALOG this means to specialize the database according to constants in both the query and subqueries.

Partial evaluation is best combined with a bottom-up evaluation strategy, because known information (constants) is moved down in the database and this optimizes the bottom-up evaluation. We have not considered partial evaluation in general because we use a top-down evaluation strategy. However, unfolding is a part of partial evaluation and we found that unfolding is independent of the query in DATALOG. This means that evaluating the unfolding of non-recursive predicates in the database is not needed whenever a new query is asked but only when the database is changed. Unfolding is further discussed in [Tamaki 84] and in [Bondorf 90].

3.7.2 Other approaches

Another optimization strategy has been considered by [Kemp 89] who consider propagating comparison predicates closer to the extensional predicates. This is done statically but we are convinced that it is possible to do that dynamically by propagating both comparison predicates and constants with the subquery. Since both constants and comparison predicates are able to perform a selection on the solutions to the subquery. The idea is to make selections on the set of solutions as early as possible.

Also [Sagiv 87] considers optimization of DATALOG programs, but the class of programs on which it would have any affect is relatively small. However, more research is needed in this area.

Chapter 4

Safety of Minimal Datalog Programs

4.1 Safety of Pure Datalog Programs

Definition of safety

> *A* DATALOG *programs is safe, if the evaluation of the query terminates with a finite set of solutions.*

Safety of pure DATALOG programs is guaranteed by demanding that rules are strongly *range restricted*. A rule is strongly range restricted if every variable of the head appears somewhere in the body. [Bancilhon 86b] concludes (in the case of pure DATALOG) that any query defined over a set of strongly range restricted rules is safe. However, this is a sufficient condition but it is not necessary. We are not interested in asking whatever queries there may be to ask, so instead we will demand that the database is range restricted with respect to the query.

The following example illustrates a pure DATALOG program that is range restricted with respect to the query (or just range restricted), but not strongly range restricted.

```
?– eq(3,X).
eq(Y,Y).
```

The variable Y in the rule does not occur in the rule body (because there is no rule body), but the variable X in the query is range restricted because of the internal binding of the arguments in the eq predicate. One solution: X=3.

4.2 The Range Restriction Demand

There are two types of infinite predicates: standard predicates and user defined recursive predicates without a fixed point. The standard predicates are base predicates, i.e. infinitely many rules which are all facts, while the user defined predicate must consist of at least one rule which is recursive through an arithmetic predicate. We use the *range restriction demand* to ensure a safe use of the standard predicates, i.e. we will not accept queries or subqueries that will cause the standard predicate to return an infinite solution. With just one exception for multiplication (see page 49).

When we say that a program is range restricted, we guarantee a safe use of all standard predicates, but not safety of the program it self, i.e. if the program terminates, then the answer is correct and finite.

4.3 Binding Patterns

In order to determine whether a program is range restricted or not, we introduce *binding patterns* for predicates. Predicates may demand that certain arguments are bound before evaluation (we call such demands for PRE-BINDINGS). On the other hand it is interesting to know whether a free variable can be guaranteed bound or not after the evaluation. (These guarantees are called POST-BINDINGS).

The combination of PRE- and POST-BINDINGS is called an *adornment*. The following table illustrates the possibilities.

PRE-BINDING	POST-BINDING	Comment
FREE	FREE	Caused by an anonymous variable.
FREE	BOUND	Facts have this adornment.
BOUND	BOUND	Comparative predicates and negated predicates demand this adornment.
BOUND	FREE	This is impossible !

We have three different adornments, which we will name after their binding pattern, POST-FREE (FF), FREE-BOUND (FB), and PRE-BOUND (BB).

The fourth adornment is impossible because a predicate can not change a bound argument to be a free one. These adornments are used below to adorn predicates.

4.3.1 Data-flow and Adornments

The adornment contains important information about the possible data-flow. The adornment of an argument indicates whether data is flowing in or out of the argument.

- FREE-BOUND (FB): indicates that this argument produces data. Data, if any, is returned through this variable argument. In PROLOG such an argument would be referred to as an output argument.
- POST-FREE (FF): is much like FREE-BOUND (FB), i.e. it may return values, but the variable argument may also return an anonymous variable and therefore still be free. An argument that is adorned POST-FREE (FF) may only return data, but we can not guarantee that it will not return an anonymous variable.
- PRE-BOUND (BB): indicates that this predicate does not produce any data. On the contrary it may reduce the set of solutions. The argument would be seen as an input parameter because data must be present.

4.4 Safety of Minimal Datalog Programs

As for pure DATALOG we would like to ensure safety of minimal DATALOG programs, but as the following example shows there might be some problems with arithmetics.

Example: This query is not safe because the predicate p inductively produces the set of natural numbers which is infinitely large.

```
?- p(X).
p(X+1) :- p(X).
p(0).
```

According to our definition of range restriction in section 4.2 this program is range restricted, but the set of solutions is infinite! Therefore we will first discuss safety of minimal DATALOG programs without arithmetics and then with arithmetics.

4.4.1 Without Arithmetics

That a variable is range restricted really means that it can not be bound to an infinite set of values. Since all base predicates consists of a finite set of facts, it is only the comparative predicates that can introduce infinite solutions. So if we can ensure that a comparative predicate can never be evaluated before all its arguments are bound, then the program is safe. This is a sufficient condition but it is not necessary, i.e. there exist programs, which depend on arithmetics, that are safe.

The binding between arguments in a rule head is a special case. If the variable occurs in the rule body with the adornment FREE-BOUND (FB), then all arguments where the variable occurs in the head can be adorned FREE-BOUND (FB), but otherwise at least one of the arguments must be PRE-BOUND (BB) (se the following table).

Example of a program with multiple occurrences of a variable in the rule head. In this program at least one of the arguments in the query eq3 must be bound.

```
?- eq3(X,7,Z).
eq3(X,X,X).
```

However, if the eq3 predicate was defined strongly range restricted, i.e. all variables in the head occurs in the body, then it is not necessary to demand at least one of the arguments to be bound.

```
?- eq3(X,Y,Z).
eq3(X,X,X) :- domain(X).
domain(7).
```

Table of adornments

Predicates	Adornments
Facts	$\langle$FB$\rangle^*$
Binding *eq3(X,X,X).*	$\langle$BB,FB,FB$\rangle$ $\langle$FB,BB,FB$\rangle$ $\langle$FB,FB,BB$\rangle$
NOT EQUAL	$\langle$BB,BB$\rangle$
LESS THAN	$\langle$BB,BB$\rangle$
LESS THAN OR EQUAL	$\langle$BB,BB$\rangle$

Example of a safe program.

```
?- p(X).
p(X) :- r(X) & X > 2.
r(1).
r(2).
r(4).
r(8).
```

Solution.

```
X=4.
X=8.
```

4.4.2 With Arithmetics

Introduction

As we have already seen, a predicate that is defined recursively through addition may not be safe. But even non-recursive programs may give problems. Consider the following program, with multiplication, and try intuitively to state whether the predicate p is range restricted or not, i.e. if the use of standard predicates is safe.

```
?- p(X).
p(X) :- r(Y) & X * Y = Y & X != Y.
r(5).
r(3).
r(0).
```

The predicate is **not** range restricted, because when Y is bound to 0 (zero), we have X ∗ 0 = 0, which is true for all numbers, then X is not BOUND and then the adornments of not-equal (!=) is violated in the last literal X != Y.

Addition

Addition is a three argument predicate and it is a base predicate because it only consists of facts, but it differs from user-defined base predicates by being infinitely large. The safe use of addition is to demand that at least two out of the tree arguments are bound. This is illustrated with the adornments in the table below.

Table of adornments

Predicates	Adornments
ADDITION	⟨FB,BB,BB⟩
Z + Y = X	⟨BB,FB,BB⟩
	⟨BB,BB,FB⟩

Example on the use of addition

```
?- p(X).
p(X) :- X + 3 = 5.
```

Solution

X = 2.

Multiplication

As for addition we would like to assume that when two arguments are bound before evaluation, then the third is guaranteed bound after, but as shown in the introductory example this is not the case for multiplication. So to ensure a safe use of all standard predicates, we give multiplication the following adornment.

Table of adornments

Predicates	Adornments
MULTIPLICATION	⟨FF,BB,BB⟩
$Z * Y = X$	⟨BB,FF,BB⟩
	⟨BB,BB,FB⟩

This adornment allows us statically to determine the range restriction demand.

Now, this is in many cases an adornment that is too strong, so one could make it up to the user to ensure that the interpreter will never have to evaluate `X * 0 = 0`, and thereby obtain a weaker adornment.

Table of adornments

Predicates	Adornments
MULTIPLICATION	⟨FB,BB,BB⟩
$Z * Y = X$	⟨BB,FB,BB⟩
	⟨BB,BB,FB⟩

But then the exception must never occur. Otherwise it will cause a run-time error.

Exception

Predicates	Adornments
Exception for multiplication	⟨FF,0,0⟩ ⟨0,FF,0⟩

Example on the use of multiplication

```
?– p(X).
p(X) :– X * 4 = 6.
```

Solution

```
X = 3/2.
```

4.4.3 Manually Range Restricting Predicates

It is not very interesting when the interpreter *goes into* an infinite loop and in order to prevent this when using arithmetics we will extend the syntax of DATALOG to enable the user to specify the *range* of predicate arguments. The idea is to specify an interval in which the predicate argument may be defined, i.e. it is not defined outside the specified range. Then when we ask queries where the argument is bound to values outside the domain range we know, without any evaluation, that the predicate will fail. This helps the interpreter to evaluate the predicate, but it does not guarantee safety of the program. In the following two examples the first is safe while the second is not, despite that the user has given a definition interval.

Fibonacci

```
?– fib(5,FIB).
fib(X,Y1+Y2) :– fib(X-1,Y1) & fib(X-2,Y2).
fib(0,1).
fib(1,1).
fib[1]= [0;10].
```

fib[1]= [0;10]. means that the first argument of the predicate fib is defined in the closed interval from 0 to 10. The result is that any query or subquery on the form ?– fib(c,Y), where c $\notin$ [0;10], will fail. In this case one can safely ask ?– fib(X,Y) and the interpreter should return the 11 solutions.

Infinite loop

```
?– p(X).
p(X) :– p(X*2).
p(1).
p[1]= [0;1].
```

In this case the interpreter would start by concluding p(1) and from that conclude p(1/2) and then p(1/4) and so on, but never p(0). However, it can be made safe by changing the range definition into:

```
p[1]= [1/4;1].
```

Then when we conclude p(1/8) it is not added to the set of solutions and therefore we can not conclude p(1/16) and the evaluation will terminate.

Backus-Naur Form

2.	$\langle DATABASE \rangle$	$::= \{ \ \langle RULE \rangle. \mid \langle RANGE \rangle. \ \}^*$
24.	$\langle RANGE \rangle$	$::= \langle ID \rangle \ [\ \langle \mathsf{N}_1 \rangle \ ] = [\ \langle NUMBER \rangle \ ; \langle NUMBER \rangle \ ]$

Abstract Domains

1.	$DATALOG$	$= QUERY \times DATABASE \times INFO$
35.	$INFO$	$= RANGE$
36.	$RANGE$	$:: ID \underset{m}{\rightarrow} \mathsf{N}_1 \underset{m}{\rightarrow} NUMBER \times NUMBER$

4.4.4 Safety Level

We propose that a deductive database system like ours should be capable of determining the level of safety for each predicate.

If a program is range restricted and without arithmetics, it is guaranteed safe (level 1). If the program is range restricted and with arithmetics it can be guaranteed that the standard predicates are used safely (level 2). Level 3 indicates that the use of multiplication is user responsible. Further levels could be introduced to experiment for example with non-stratified programs, but we have not investigated this further.

We propose that the user indicates the level of safety when needed. Level 2 is default, so if the user want a guarantee that a given predicate is safe, he/she just demand it to be of level 1 and the system should check whether this is possible or not.

For a predicate to be of level 1 all predicates on which it depends must also be of level 1. For a predicate to be of level 2 all predicates on which it depends must also be of level 1 or 2. Any predicate is at least of level 3.

Backus-Naur Form

2.	$\langle DATABASE \rangle$	$::= \{\ \langle RULE \rangle.\ \mid \langle RANGE \rangle.\ \mid \langle SAFETY \rangle.\ \}^*$
25.	$\langle SAFETY \rangle$	$::= \langle ID \rangle\ :\ \langle NUMBER \rangle$

Abstract Domains

35.	$INFO$	$= RANGE \times SAFETY$
37.	$SAFETY$	$::\ ID \underset{m}{\Rightarrow} NUMBER$

4.5 Summary

In this chapter we have discussed *safety* and the weaker property *range restriction* of minimal DATALOG programs. When programs are range restricted, we guarantee a safe use of all standard predicates but we do not guarantee termination of the query answering algorithm. Predicates that furthermore do not depend on arithmetical predicates can be guaranteed to terminate and are thereby safe.

By introducing binding patterns for predicates we have determined the adornments of standard predicates and facts.

We have extended the syntax of DATALOG further. The user may now manually specify:

- the domain of predicate arguments, and
- the level of safety of a predicate.

Chapter 5

Data-flow Analysis

To determine whether or not there exist a way to evaluate the query without violating the demanded pre-bindings for the standard predicates (comparative and arithmetical predicates) and negated predicates we introduce a method to calculate the set of allowed queries and then test if the actual query is a member of this set.

5.1 Binding Patterns for Rules

We have already determined the possible binding patterns for facts, comparative predicates, and arithmetical predicates and now we give a method to statically determine the possible binding patterns for each user defined predicate. To do so we use the rule-graph introduced in section 3.5.4.

To represent the adornment information we use the following auxiliary domains.

Auxiliary Domains

1.	*ADORNMENT*	$= ID \underset{m}{\rightarrow} PATTERN\text{-}\underline{\textbf{set}}$
2.	*PATTERN*	$= BINDING^*$
3.	*BINDING*	= POST-FREE | FREE-BOUND | PRE-BOUND

The main advantage of adorning the predicates in the order given by the rule-graph is that all adornments of the literals in the body of any rule will always by known. Therefore we can adorn the database in a bottom-up manner.

5.2 Adorning a Rule

To adorn a rule means to determine whether a variable argument can be FREE or is demanded to be BOUND before evaluation, and if it may be FREE then to determine whether it is guaranteed to be bound after the evaluation or it is left un-bound, i.e. FREE.

There are three possible terms that can occur in the head of a minimal DATALOG rule: a constant, an anonymous variable, or a variable.

- Constants: The adornment is the same as for facts, i.e. FREE-BOUND (FB). If one asks a constant argument with a free variable, then the variable is bound to the constant. If the variable is bound then its value is compared with the constant.

  Example of a constant in the rule header

  ```
  ?– p(X).
  p(a).
  ```

  Solution

  ```
  X = a.
  ```

- Anonymous variables: The adornment is POST-FREE (FF). If one ask an anonymous variable with a free variable then the variable is temporary bound to the anonymous variable (as in PROLOG). Any value will match an anonymous variable, i.e. bound variables or constant values.

  Example of an anonymous variable in the rule header

  ```
  ?– p(X).
  p(_).
  ```

  Solution

  ```
  X = _.
  ```

  Note that returning an anonymous variable is quite normal in logic programming languages like e.g. PROLOG.

- Variables can not be adorned in a simple fashion, in the worst case it takes an extensive investigation of all possible combinations of pre-bindings of the variables in the head combined with all permutations of the literals in the body. However, this worst case can be avoided in most cases and we show how this can be done.

Example

```
r(X,Y) :– p(X,Y) & q(X,Y).
r(X,X).
```

where predicate p is adorned with (BB,FB) and predicate q is adorned with (FB,BB). By permuting the literals, testing different pre-binding patterns, and proving the corresponding post-binding, we are able to determine the possible adornment of predicate r. In this example there are only two possibilities of selecting a literal from the rule body.

Possible adornments

```
r{bb,fb}.
r{fb,bb}.
```

This means that one of the arguments must be bound and the other may be free, before a query to r can be asked.

5.2.1 The Algorithm

Class Dividing the Rule Body

The worst problem of finding all possible adornments is that we maybe have to test all possible permutations of the literals in the body. However, it is possible to divide the body into three classes according to their adornments. Literals in the first class must not be adorned with PRE-BOUND (BB), literals in the third class must not be adorned with FREE-BOUND (FB), and the second class contains the rest.

Where do the standard and base predicates belong?

first class: base predicates, i.e. all rules are facts.
second class: arithmetic predicates.
third class: comparative predicates and negated predicates.

The result of this class division is that only literals in the second class needs to be permuted in order to find the possible adornments of the rule. The cost of performing a permutation is $O(n!)$ where n is the number of literals in the second class.

Using information from other rules

If we know that an argument is demanded by another rule to be PRE-BOUND (BB), then there is no need to investigate any other possibilities.

Adornments of first class literals

All variable in the head that also occur in literals in the first class and that here is adorned FREE-BOUND (FB) can be adorned FB without further investigations.

Adorning the rest of the variables

For the rest of the variables we just have to try all possibilities, i.e. investigate as many PRE-BINDING combinations as needed and see which POST-BINDINGS we can prove.

- All pre-bindings free. First we pre-adorn all variables to be free and if this adornment can be proved with a corresponding post-adornment, then we do not need any further investigations.
- All combinations of free and bound. Otherwise we have to consider all combinations of pre-adornments of the variables and try to prove each of them and their corresponding post-adornments.
- No possible adornments. If this does not result in any possible adornments of the rule, then their is an error, i.e. the rule can not be proved range restricted, and the evaluation stops.

Proving an Adornment

When proving an adornment we investigate the first and second class literals in the rule body for their ability to binding variables and then we control that all variables in literals in the third class are bound by either the literals in the two first classes or by the variables' occurrence in the head.

5.3 Adorning Predicates

In order to determine the adornment of a predicate, we have to compare the adornments of the corresponding rules. The strong adornment, PRE-BOUND (BB), must be satisfied, and a predicate argument that can not be guarantied FREE-BOUND (FB) in just one rule is adorned with a POST-FREE (FF). The following table illustrates these relations.

5.3.1 Comparing Possible Adornments

One rule	Another rule	Predicate
PRE-BOUND	PRE-BOUND	PRE-BOUND
PRE-BOUND	POST-FREE	PRE-BOUND
PRE-BOUND	FREE-BOUND	PRE-BOUND
POST-FREE	POST-FREE	POST-FREE
POST-FREE	FREE-BOUND	POST-FREE
FREE-BOUND	FREE-BOUND	FREE-BOUND

Example of different adornments

```
?– p(5).
p(2).              p{ fb}.
p(_).              p{ ff}.
p(X) :– X < 3.     p{ bb}.
```

Each rule may have several possible adornments, so it is necessary to compare all combinations according to the table above. The adornment of p is PRE-BOUND (BB).

5.3.2 The Algorithm

Predicates with facts in the extensional database are all initially adorned with FREE-BOUND (FB). Predicate names that occur on the right hand side of rules in the database, but never on the left hand side, are assumed to be base predicates without any solutions, i.e. they belong to the extensional database and are therefore adorned as facts. The arity of these predicates are known from the aritycheck in section 2.4.4.

From the rule-graph we have information on what rules that can be adorned in one step and what rules that must be adorned repeatedly until reaching a fixed point. When adorning a recursive predicate the adornment may change during the iteration. A consequence of this is that when we, during the evaluation of the adornment of a rule, divide the body of the rule into three classes, the contents of these classes may change because arguments in recursive predicates in the body may be demanded to be PRE-BOUND (BB). This is the argument that the recursion will stop, because when a predicate argument is once adorned

PRE-BOUND (BB), then this can not be changed (see the table above).

Example

In this example we show how the algorithm works step by step.

query ?– p(3,5).

rule 1 p(_,3).

rule 2 p(X,Y) :– X < 10 & p(Y,X).

Adorning predicates

step p(_,3) $\Longrightarrow$ p{ ff,fb}

rep.1 p(X,Y) :– p(Y,X) & X < 10. $\Longrightarrow$ p{ ff,ff}

rep.2 p(X,Y) :– p(Y,X) & X < 10. $\Longrightarrow$ p{ bb,ff}

rep.3 p(X,Y) :– p(Y,X) & X < 10. $\Longrightarrow$ p{ bb,bb}

stop p(X,Y) :– p(Y,X) & X < 10. $\Longrightarrow$ p{ bb,bb}

In the *step* the adornment of *rule 1* is evaluated. The first argument of the literal p in *rule 2* is adorned with (FF) and since it occurs as second argument in the rule header, which is also p, the adornment of p is now: p{ff,ff} , but since the comparative predicate < demands that its arguments must be guaranteed bound before evaluation the adornment of p is demanded to be: p{bb,bb} , which is a fixed point and the evaluation was terminated.

5.4 Manually Adorning Predicates

Since it may be quite resource demanding to calculate all possible adornments for a predicate one may wish to adorn some of the predicates manually. We extend the syntax in the following way:

Backus-Naur Form

2.	$\langle DATABASE \rangle$	::= { $\langle RULE \rangle$. | $\langle RANGE \rangle$. | $\langle SAFETY \rangle$.
.1		| $\langle ADORN \rangle$. }*
26.	$\langle ADORN \rangle$	::= $\langle ID \rangle$ { $\langle BP \rangle${ , $\langle BP \rangle$ }* }
27.	$\langle BP \rangle$	::= fb | ff | bb

Abstract Domains

35.	*INFO*	$= RANGE \times SAFETY \times ADORN$
38.	*ADORN*	$::$ *ID* $\underset{m}{\rightarrow}$ *(BP*$^+$*)***-set**
39.	*BP*	= FREE-BOUND | POST-FREE | PRE-BOUND

Example

```
?– eq(5,X).
eq(X,X).
eq{bb,fb}.
```

The main advantage of manually adorning predicates is that the system does not have to find all possible adornments, but only to check whether the user given adornments are possible.

5.5 Summary

In this chapter we have developed a method to determine if a given program can be evaluated without violating the pre-binding demands of comparative and arithmetical predicates.

And we have extended the syntax of DATALOG further. The user may now manually specify:

- the adornment of a predicate.

With these extra facilities the user now has a powerful deductive database language with an extended control of the database well-formedness.

5.5.1 Comments on the Data-flow analysis

Our data-flow analysis is very different from the data-flow analysis normally performed on PROLOG programs, which is a top-down analysis which depend on the goal [Jensen 84]. Our data-flow analysis is a bottom-up strategy which investigates the possible data-flow for each predicate in the database, i.e. all legal uses of a predicate is determined. Finally the binding pattern of the query is compared with the information on the corresponding predicate. The possible data-flow is represented by adornments.

Chapter 6

Query Answering Strategies

6.1 Classification of the Strategies

To classify the different query answering strategies presented in the literature, [Bancilhon 86b] defines each strategy by an application domain (i.e. a class of rules for which it applies) and an algorithm for replying to queries given such a set of rules.

Query Evaluation vs. Query Optimization

There are two types of approaches in the literature: First, actual query answering methods which will produce the answer to the query and second, term rewriting systems which will optimize a given program, but which must be combined with an evaluation method to answer the query.

Further Method Classification

The answering methods can by classified further. The literature distinguishes between *interpreted* and *compiled* methods, between *recursive* and *iterative* methods, their capability to focus on the *potentially relevant facts*, and the direction of data-flow, i.e. *Top Down* or *Bottom Up*.

Definition of linear rule

> *A rule is linear if it is recursive, and the recursive predicate appears once and only once in the body. A recursive rule is otherwise non-linear.*

Some of the methods below can not handle non-linear reclusive rules, i.e. their application range is linear recursive rules.

6.2 Some of the Methods

Each method is briefly described by its application range, and its origin.

Jacobi

The Jacobi algorithm is presented in [Ceri 90, page 146] and is a member of the class of methods called *Naive evaluation methods.* It is a bottom-up, compiled, iterative strategy. There have been several approaches to this strategy and further references can be found in [Bancilhon 86b]. All rules must be *strongly range restricted* (see section 4.1). However, it is possible to introduce standard predicates like comparison and arithmetics. As we shall see in the performance study this method is relatively slow.

Semi-Naive

Another class of bottom-up strategies is the *Semi-Naive evaluation methods.* These methods use the same approach as naive evaluation, but tries to cut down on the number of duplications. The basic principle of the semi-naive method is the evaluation of a *differential* of a rule body. The differential is a first order formula which should compute only the new tuples. However, the problem is automatically to generate this differential. According to [Bancilhon 86b]: “This problem is not solved in its entirety and only a number of transformations are known”. The application range of the method (if it works) is the same as for Jacobi (see above).

Magic Sets

The *Magic Sets* algorithm was first introduced informally in [Bancilhon 86a] and has since then received much attention. Some of the most interesting approaches are [Mumick 90, Beeri 87, Ullman 89b]. The most easy-to-understand presentation of Magic Sets can be found in [Ceri 90]. Magic Sets is a rewriting strategy. It changes existing rules and introduces new ones, the result being about twice as many rules.

The method is based on the idea of *sideways information passing.* Intuitively, given a certain rule and a subgoal in the rule body with some bound arguments, one can solve this subgoal, and so obtain bindings for uninstantiated variables in other argument positions. These bindings can be transferred to other subgoals in the rule body, and they, in their turn, transmit bindings to other variables. A behavior like this is common for top-down resolutions-based methods.

The application range is in general all DATALOG rules, but non-linear recursive programs are not in general optimized by the Magic Sets strategy.

Counting

Counting is derived from Magic Sets and it applies under two conditions: (i) the data is *acyclic* and (ii) there is at most one recursive rule for each predicate, and it is linear. It was first presented in [Bancilhon 86a] and later improved in [Beeri 87] to the Supplementary Counting Method. Since the method is very specialized and can only be used as an alternative to Magic Sets when the conditions are fulfilled. [Sacca 87] introduces a method which automatically decides whether to apply the Counting method or the Magic Sets method.

Recursive Query Subquery

QSQR is a top-down interpreted recursive strategy. QSQR is based on SLD-resolution, but differs from PROLOG by computing sets of answer-tuples instead of just one tuple at a time and by saving intermediate results for possible reuse. The QSQR algorithm apply under all conditions. The termination of recursive calls is done by testing for new solutions and for new subgoals. Its original application range was *Strongly range restricted* rules and no arithmetic. QSQR was introduces by Laurent Vieille in 1986, but the algorithm was found to be incomplete. A complete version can be found in both [Vieille 89] and [Ceri 90].

Partial Evaluation

[Bondorf 90] consider specialization of PROLOG programs without *cut*. The idea is that literals/subgoals with constant terms are specialized and that bindings between terms in predicates like eq(X,X) lead to *sideways information passing* where the predicate occurs in the database.

Illustration

$$mix(p, q, p_q)$$

The residual program p_q is the result of a specialization of the program p with respect to query q. mix is the partial evaluator.

Partial Evaluation can be combined with a bottom-up evaluation strategy like Naive or Semi-Naive evaluation. See section 3.7.1 for more information on partial evaluation.

Extensions to the methods

All evaluation methods are designed to perform a fixed point computation for a subset of pure DATALOG. By demanding programs to be stratified most methods can be extended to handle negation. Most algorithms can handle comparison predicates, but not equally efficient. Anonymous variables should not cause any trouble, but arithmetical predicates could cause unforeseen problems.

6.3 Performance

The only really benchmarks on query answering methods for DATALOG can be found in [Bancilhon 86b]. We present the most important results in the table below. They use the symbol "<<" to denote the fact that a method performs an order of magnitude better than the other. Methods of same order of magnitude are separated by a comma, such that the best methods are placed to the left. The programs used as benchmarks are in appendix A.

Method Performance Study

- Linear recursive *ancestor* example. See page 98.
  C << MS,QSQR << SN << J

- Non linear recursive *ancestor* example. See page 99.
  QSQR << SN,MS << J

- Linear recursive *same generation* example. See page 101.
  C << MS,QSQR << SN << J

Abbreviations: C=Counting; MS=Magic Sets; QSQR=Recursive Query-Subquery; SN=Semi-Naive; J=Jacobi. Several methods have not been considered because they always performs relatively worse than the methods shown here. More information on those can be found in [Bancilhon 86b, Ceri 90, Ullman 89a].

The performance study shows each method's ability to focus on the relevant data, i.e. it ability cut irrelevant work down to a minimum.

Bottom-Up vs. Top-Down

The best bottom-up strategy and the best top-down strategy are presented here.

- Bottom-Up: Use the Counting rewriting method when possible. Otherwise use Magic Sets rewriting. The rewritten program is evaluated by Semi-Naive evaluation.

- Top-Down: The Recursive Query Subquery resolution method.

Both strategies could be improved by a preceding data-flow analysis. In the performance study, Counting and Magic Sets performs better than QSQR in the linear case, but orders of magnitude worse in the non-linear case. A combination of several methods would probably give the optimal result.

6.4 Choice of method

Because of the weak presentation on how to automatically compute the *differential* of a rule body we do not have faith in the Semi-Naive evaluation method, and because of the possibility of introducing a powerful *selection function* based on the results we obtained from the data-flow analysis we have decided to work with the QSQR algorithm. QSQR perform best on non-linear predicates which is important when building DATALOG interpreters in DATALOG [JFN 91a].

6.5 Summary

In this chapter we briefly comment on some of the most famous query answering algorithms. We briefly describ the application range for each method, and its origin. Further we compare the best bottom-up vs. top-down strategy and justify our choice of top-down algorithm QSQR.

Another Approach

Another approach in comparing methods is [Ullman 89b]. Jeffrey Ullman succeeds in proving that Semi-Naive evaluation combined with the Magic Sets rewriting performs better than his own Top-Down resolution method called *Queue-Based Rule/Goal Tree expansion.* However, this is only valid in the case of linear rules, because Magic Sets would not improve Semi-Naive evaluation in the case of non-linear rules [Bancilhon 86b].

Comment

Some of the papers claim that most real-life DATALOG recursive programs are linear recursive, but the *meta-datalog interpreter* example [JFN 91a] on page 95 disproves this. Interpreting DATALOG in DATALOG can only be done by non-linear recursive rules.

Chapter 7

Interpreter for Minimal Datalog Programs

7.1 Introduction

7.1.1 Source of Inspiration

Our work and ideas in this chapter are based on the results by [Vieille 89] and the overview of resolution techniques in [Ceri 90]. These results are used to develop a top-down resolution method for *minimal* DATALOG.

7.1.2 "Foundations of Datalog" by Ceri et al.

Among other subjects [Ceri 90] defines the logical semantics of pure DATALOG programs by using the concepts of *model theory*. This allows them to define the concepts of *logical truth* and *logical consequence*. They explain important notions, such as *Herbrand base*, *substitution*, and *unification*.

[Ceri 90] also develops the *proof theory* of pure DATALOG and demonstrates that the method is both *sound* and *complete*. Further they give an introduction to *resolution* and *backward chaining*. They also present a compleate version of the QSQR algorithm.

7.1.3 "Recursive Query Processing" by Vieille.

Vieille gives a brief introduction to SLD resolution and obtains the following result.

Query Answering Procedure = Search Space + Search Techniques

He introduces two types of techniques to enrich SLD resolution. First the AL-technique, standing for *Admissibility test* and *Lemma resolution*, which tests whether the current subgoal is *new* (admissibility test), and, in case it is not, re-uses the answers obtained in its previous occurrence (lemma resolution). The second technique aims at *pruning redundant parts* of SLD and SLD-AL trees, i.e. subtrees which contain answers that are contained by another part of the tree. Further he discusses (db-)subsumption, and *local and global* optimization of SLD-AL resolution. The results are used in his application to deductive databases: *the QoSaQ procedure* = Q_S_Q + glObAl optimization.

7.1.4 SLD-resolution, Prolog and Datalog

The main differences in interpreting PROLOG and DATALOG programs with SLD-resolution are: 1) the selection function, 2) the unification and 3) returning values.

Selection Function

In PROLOG the Selection Function just selects the next literal in the body (a list of rules) while in DATALOG the Selection Function must consider all literals in the body (a set of rules).

Most General Unifier

Since PROLOG terms may be predicate symbols it may be difficult to compute the *most general unifier* while it is almost trivial in DATALOG.

Returning values

The counterpart to the DATALOG query is the PROLOG goal. In PROLOG the interpreter returns one tuple at a time to the goal while in DATALOG the whole set of tuples is returned to the query.

7.1.5 Semantics

Until now we have not discussed the semantics of DATALOG. There are several ways to specify the semantics or the meaning of a program written in a logic language.

Mathematical Logic

When consulting a DATALOG database with a query q we are really asking whether q is true or not. In Mathematical Logic two approaches to the definition of truth exist: the *model-theoretic* approach and the *proof-theoretic* approach. Roughly speaking, in Model Theory a logical sentence is (tautologically) true if and only if it is true in all possible situations, or in all *possible worlds*. On the other hand, in Proof Theory a sentence is true if and only if it can be derived by the application of some given rules from a given set of axioms.

Approaches to a formal definition of the semantics of pure DATALOG or *Horn clauses* can be found in [Ceri 90, JFN 89, Nilsson 90, Ullman 89a, Van Emden 76].

Operational Semantics

When extending DATALOG with extra facilities it gets more difficult to use the mathematical logic. We will instead give a more programming like definition of the semantics of DATALOG. This is often called an *Operational Semantics.* Operational Semantics of pure DATALOG can be found in [Ceri 90, Ullman 89a, Vieille 89]. Term rewriting systems are not part of the semantics and are therefore not referred to here, see instead chapter 6.

Our operational semantics of minimal DATALOG will be the specification of the algorithm *Recursive Query Subquery.* The algorithm has its origin in [Ceri 90] and we extend it further to handle both the extra predicates and the manually domain-restriction of bound arguments that we introduce.

SLD-resolution

SLD-resolution is an abbreviation for *Linear Resolution with Selection Function for Definite Clauses* and introductions to this strategy can be found in [Ceri 90, JFN 89, Nilsson 90, Ullman 89a, Vieille 89].

Extensions of SLD-resolution such as SLDNF and SLD-AL are presented in [Nilsson 90, Vieille 89].

7.2 Domains and Conditions

To represent the query and subqueries we introduce the *Generalized Query* (GQ), from [Ceri 90], however it is not quite the same. The top query is only capable of having one value per argument, i.e. a constant, while the GQ is cabable of having finitly many. The *answer* to a generalized query is represented in the same way as the GQ. We represent the answer and the generalized query as a *relation* [Ullman 89a]. In META-IV domains it looks like this:

Abstract Domains

1.	*G-QUERY*	$= R$
2.	*ANSWER*	$= R$
3.	*R*	$:: ID \times TERMLIST \times TSS$
4.	*TSS*	$=$ *TERMLIST*<u>**-set**</u>
5.	*TERMLIST*	$= TERM^*$
6.	*TERM*	$= VAR \mid CON \mid$ <u>ANONYMOUS VARIABLE</u>

The Relation

A relation consists of its name (*ID*), the termlist (*TERMLIST*), and the set of tuples (*TSS*). The tuples in *TSS* and the termlist are of equal length and the termlist forms a mask on the tuples, i.e. each position in the termlist corresponds to the same position in every tuple. If a position in the termlist contains a constant, then that constant is also in that position in every tuple. If a position in the termlist contains either a variable or an anonymous variable, then may that position in any tuple contain either an arbitrary constant (which indicate that the variable is bound) or an anonymous variable (which indicate that the variable is free).

Generalized Query

The generalized query is normaly produced from a literal and may therefore contain dublicate variables, anonymous variables and constants in the termlist.

Answer Relation

The answer relations is produced from a rule and the termlist contains one occurence of each variable occuring in the rule.

7.3 Interpreter

Transform the query into a generalized query.

```
7.     interpreter(query) ≜
 .1       let generalized-query = generalize(query) in
 .2          qsqr(generalized-query)
7.3    type: interpreter : QUERY → ANSWER
```

Annotations to *interpreter*:

.1 Transform the query into a generalized query.

.2 Call Recursive Query Subquery with this query.

End of annotations

7.4 Recursive Query Subquery

Our algorithm is based on the *recursive query subquery* algorithm presented in [Ceri 90, pp. 155-160]. In the following specification only the important high-level functions are specified, the rest are just explaind in the annotation. A more compleate specification is presented in appendix D.

In the following specification of the algorithm the DATALOG database is accessible from an external database as *database*.

QSQR

```
8.    $qsqr(query_1) \triangleq$
 .1     let $query_2 = within\text{-}range(query_1)$ in
 .2        let $answer = \mathsf{Y}\ union\{\ consult\text{-}rule(query_2,rule)\ |$
 .3            $rule \in database \wedge match(query_2,rule)\}$ in
 .4          $within\text{-}range(answer)$
8.5   type: $qsqr : G\text{-}QUERY \to ANSWER$
```

Annotations to *qsqr*:

.1 If a domain is specified for the predicate, then *within-range* takes out those query-tuples that are not within the specified domain.

.2 Find the fixed point of unioning the answers of consulting all rules that matches the query.

.3 Answer-tuples that are not within the specified domain are removed.

End of annotations

Readers that are not familiar with the Y notaion are referred to Chapter 8. The Lambda Calulus, Section 11. The fixed point finding operator in [Bjørner 89, Vol. I].

consult-rule

```
9.    $consult\text{-}rule(query_1,rule_1) \triangleq$
 .1     let $query_2 = rename(query_1,rule_1)$ in
 .2     let $query_3 = filter(query_2,rule_1)$ in
 .3     let $\theta = unify(query_3,rule_1)$ in
 .4        if $\theta \neq \bigtriangledown$
 .5        then let $rule_2 = \mathcal{S}\text{-}RULE(rule_1,\theta)$ in
 .6             let $answer_1 = pre\text{-}loop(query_3,rule_2)$ in
 .7             let mk-$RULE(head,body) = rule_2$ in
 .8             let $answer_2 = loop(answer_1,body)$ in
 .9                $project(answer_2,head)$
 .10       else $fail(query_1)$
9.11  type: $consult\text{-}rule : G\text{-}QUERY \times RULE \to ANSWER$
```

Annotations to *consult-rule*:

.1 Before unifying the query with the rule we must ensure that the query and the rule header do not have variables in common. This is done by renaming variables in the query.

.2 Then filter out those tuples in the generalized query that do not unify with the rule.

.3 Try to compute the *most general unifier* (**mgu**) for query and the rule.

.4 − .5 If the **mgu** θ exists ($\bigtriangledown$ means that the query and the rule header can not be unified) then substitute the rule with θ to correct for local bindings in the query, i.e. queries like ?− p(Z,Z). to rules like p(X,Y) :− r(X,Y)..

.6 *pre-loop* forms a new answer-relation, where all variables in the rule are within. It is important that all variables are present in the generalized query, for the arithmetical operations to give uninstantiated variables a value. Because it is done as a selection instead of a join.

.8 The *loop* function will return the answer-relation instantiated with the values from the body.

.9 The answer tuples are projected on to the rule head.

.10 If the query and the rule do not unify, i.e. a **mgu** does not exist, the return the fail relation.

End of annotations

loop

```
10.    loop(answer_1,body_1) ≜
  .1     if body = -set }
  .2     then anwser_1
  .3     else let bindings = binding-info(answer_1) in
  .4            let (literal,body_2) = selection-function(bindings,body_1) in
  .5            let answer_2 = cases-literal(literal,answer_1) in
  .6              loop(answer_2,body)
10.7   type: loop : ANSWER × BODY → ANSWER
```

Annotations to *loop*:

.1 − .2 When there are no more literals left stop the *loop* and return the answer relation.

.3 The information on what variables that are instantiated can be determined in two ways: one way is by looking at the adornment for each literal that have been selected and from that determine what variables that can be *guaranteed* bound and what variables that can not be *guaranteed* bound, another way is to actually check all variables in the relation to see whether they are bound or free. We have done it the later way, because variables that can not be *guaranteed* bound may actually be bound any way and thereby enable a larger set of literals to be ready for selection!

.4 Then select a literal/subgoal

.5 for execution

.6 and continue the *loop*.

End of annotations

cases-literal

```
11.    cases-literal(literal,$answer_1$) $\triangleq$
  .1     cases literal :
  .2       mk-POS(atom) $\rightarrow$
  .3         cases atom :
  .4           mk-HEAD() $\rightarrow$
  .5             let subquery = form-query($answer_1$,atom) in
  .6               let $answer_2$ = qsqr(subquery) in
  .7                 join($answer_1$,$answer_2$),
  .8           T               $\rightarrow$
  .9             selection($answer_1$,atom),
  .10      mk-NEG(atom) $\rightarrow$
  .11        let subquery = form-query($answer_1$,atom) in
  .12          let $answer_2$ = qsqr(subquery) in
  .13            difference($answer_1$,$answer_2$)
11.14  type: cases-literal : LITERAL $\times$ ANSWER $\rightarrow$ ANSWER
```

Annotations to *cases-literal*:

.1 The literal is either a positive or a negative atom.

.2 If it is positive it is either a user-predicate or a standard-predicate.

.3 − .7 If it is a user predicate the call the *recursive query subquery* algorithm recursively.

.8 − .11 Otherwise let the internal standard predicates perform a selection on the relation.

.12 − .15 In the negative call we say that all tuples that can be proved are subtracted from the relation by the difference operation. Contrary to the positive call where all proved tuples were added by the join operation.

End of annotations

7.5 Unification

An important step when evaluating logical rules top-down is *unification* between the query and a corresponding rule. To explain this unification we introduces the terms *substitution*, *unifier*, and *most general unifier*. A more general introduction can be found in both [Nilsson 90] and [Ullman 89a].

7.5.1 Substitution

A substitution θ is a finite set of bindings and each binding consists of a variable and a term. The idea is that in whatever we are substituting with θ every occurrence of the variable is substituted with the term.

Abstract Domains

1.	*SUBSTITUTION*	$=$ *BINDING**
2.	*BINDING*	$::$ *VAR* $\times$ *TERM*
3.	*TERM*	$=$ *VAR* $\mid$ *CON* $\mid$ <u>ANONYMOUS VARIABLE</u>

We need to define two basic functions; one to substitute a term and one to compose two substitutions. These two functions are used to define both the most general unifier and substitution of terms in *Termlists* and *Literals* in *Bodies.* The most general unifier is specified in this section while the other functions can be found in section D.12.

```
1.    $\mathcal{S}$-TERM$(t,\theta) \triangleq$
 .1      mk-BINDING$(t,t_i) \in \theta \rightarrow t_i$,
 .2      $\top$                          $\rightarrow t$
1.3   type: $\mathcal{S}$-TERM : TERM $\times$ SUBSTITUTION $\rightarrow$ TERM
```

Annotations to $\mathcal{S}$-*TERM*:

.1 If the term t occurs in θ then substitute it with its substitute

.2 else leave it unchanged.

End of annotations

```
2.     composition(θ,σ) ≜
 .1       θ = { } → σ,
 .2       T        →
 .3          let b ∈ θ in
 .4             let mk-BINDING(var,term_1) = b in
 .5                let term_2 = S-TERM(term_1,σ) in
 .6                   { mk-BINDING(var,term2) } ∪ composition(θ \ { b },σ)
2.7    type: composition :  SUBSTITUTION × SUBSTITUTION →
 .8                            SUBSTITUTION
```

Annotations to *composition*:

.0 Composition of two substitutions.

.1 − .6 The composite substitution consists of all bindings in θ substituted with σ and σ.

End of annotations

7.5.2 Unifier

A rule *corresponds* to the query if their names are identical. Then we know, from the arity check in section 2.4.4, that the arities are identical too. Now, if for the two termlists *QTL* and *RTL* a substitution θ exists, such that $\mathcal{S}$-*TERMLIST(RTL,θ)* = $\mathcal{S}$-*TERMLIST(QTL,θ)*, then we say that *QTL* and *RTL* are *unifiable* and the substitution θ is called a *unifier*. Of course, for a pair of termlists, several different unifiers may exist. $\mathcal{S}$-*TERMLIST* is specified in section D.12.

Important note: Since the variables of the rule are "local", we shall assume that there are no variables in common between the two termlists. If they have variables in common we will rename the variables of the query termlist (*QTL*) for the following algorithms to work right. This assumption is a pre-condition for all specifications of substitution functions.

7.5.3 Most General Unifier

Let θ and γ be substitutions. We say that θ is *more general* than γ iff a substitution λ exists such that *composition(θ,λ)* = γ. Let *QTL* and *RTL* be two termlists; a *most general unifier* (**mgu**) of *QTL* and *RTL* is a unifier which is more general than any other unifier.

The **mgu** for *QTL* and *RTL* is constructed by dynamically composing the partial substitutions which are generated step by step during the unification of the terms in the corresponding argument positions. The essential part of this **mgu** function was found in [Ceri 90]. We have formally specified the **mgu** function in the notion of META-IV and extended it to handle anonymous variables.

```
3.    $mgu(\theta,rts,qts) \triangleq$
 .1     $rts = <> \wedge qts = <> \rightarrow \theta,$
 .2     $\mathsf{T} \rightarrow$
 .3       $\underline{\textbf{let}}\ rt = \mathcal{S}\text{-}TERM(\underline{\textbf{hd}}rts,\theta),$
 .4           $qt = \mathcal{S}\text{-}TERM(\underline{\textbf{hd}}qts,\theta)\ \underline{\textbf{in}}$
 .5         $(rt = qt) \rightarrow mgu(\theta,\underline{\textbf{tl}}rts,qts),$
 .6         $\mathsf{T} \rightarrow$
 .7           $\underline{\textbf{cases}}\ qt :$
 .8             $\underline{\textbf{mk-}}VAR() \rightarrow$
 .9               $\underline{\textbf{let}}\ \sigma = composition(\theta,\{\underline{\textbf{mk-}}BINDING(qt,rt)\})\ \underline{\textbf{in}}$
 .10                $mgu(\sigma,\underline{\textbf{tl}}rts,qts),$
 .11            $\underline{\text{ANONYNOUSVARIABLE}} \rightarrow mgu(\theta,\underline{\textbf{tl}}rts,qts),$
 .12            $\mathsf{T} \rightarrow$
 .13              $\underline{\textbf{cases}}\ rt :$
 .14                $\underline{\textbf{mk-}}VAR() \rightarrow$
 .15                  $\underline{\textbf{let}}\ \sigma = composition(\theta,\{\underline{\textbf{mk-}}BINDING(rt,qt)\})\ \underline{\textbf{in}}$
 .16                    $mgu(\sigma,\underline{\textbf{tl}}rts,qts),$
 .17                $\underline{\text{ANONYNOUSVARIABLE}} \rightarrow mgu(\theta,\underline{\textbf{tl}}rts,qts),$
 .18                $\mathsf{T} \rightarrow \nabla$
3.19  $\underline{\textbf{type}}: mgu : SUBSTITUTION \times TERMLIST \times TERMLIST \rightarrow$
 .20                 $SUBSTITUTION \mid \nabla$
```

Annotations to *mgu*:

.0 The use of **mgu** is: $\theta = mgu(\{\ \},rts,qts)$. Note that the variables of $\mathcal{S}$-*TERM(qts,θ)* are those of *rts*.

End of annotations

7.5.4 Example

Consider the rule head p(X,Z,a,U) and the query p(Y,Y,V,W). The name and arity of the rule head and the query are identical and the termlists are *rts* = <*X,Z,a,U*> and *qts* = <*Y,Y,V,W*>. The result of the function call $\theta = mgu(\{\ \},rts,qts)$ is $\theta = \{bind(Y,Z),\ bind(X,Z),\ bind(V,a),\ bind(W,U)\}$. Note that $\mathcal{S}$-TERMLIST(rts,θ) = $\mathcal{S}$-TERMLIST(qts,θ) = <Z,Z,a,U >.

7.6 Selection Function

The Selection Function chooses the next subgoal of the rule body to be analyzed by the algorithm. Different selection functions will yield different kinds of optimization. The Selection Function plays an important role in optimizing the query answering procedure. The heuristic for our function is described. The selection function is formally specified in section D.11.

7.6.1 Selectable literals

Not all literals may be ready to be selected. The binding patterns must be obeyed, i.e. any literal argument that is demanded to be bound before evaluation must be bound for the literal to become selectable. The binding pattern information is described in section 4.3. The first/next literal is to be found among the selectable literals. If there are more than one, then our heuristic function will decide which one to select.

7.6.2 Most instantiated literals

[Vieille 89]'s strategy is to focus on the relevant data. This corresponds to the heuristic "make selections first". In terms of SLD resolution, this corresponds to a selection function that tries to choose the most instantiated literal. However, he never defines what he precisely means by *most instantiated.* We therefore give the following definition which we find intuitively correct.

degrees of instantiated

Before we select a literal from the rule body we have the following information on each selectable literal that is left to select: 1) predicate arity (a), 2) number of bound arguments (b), and 3) the maximal possible solutions, i.e. the cardinality of the relation to each predicate (c), the cardinality of arithmetic predicates and comparative predicates is per definition set to 1.

We define the degree of instantiated β as follows:

$\beta = b \ / \ (a * c)$, for $a \neq 0$ and $c \neq 0$
$\beta = 1$, for $a = 0$ or $c = 0$

Where ($a = 0$) means that the literal is a 0-ary predicate and where ($c = 0$) means that the predicate will fail. Literals with no instantiated variables always have a β equal to zero. Literals with all arguments bound and with only one possible solution have a β equal to one. The domain of β is [0;1], where 0 means not instantiated at all and 1 means hard instantiated. As a result: If we have to choose between to literals with all arguments bound we will choose the one with least possible solutions which we expect to make a more tough selection on the solutions already found.

So we can conclude that the closer β is to 1 the more instantiated is the literal. Our Selection Function will select the literal with the largest β.

7.7 Summary

We begin this chapter by revealing our source of inspiration, we then give a brief introduction to the subjects dealt with in [Ceri 90] and [Vieille 89] and we briefly discuss the possibilities of defining the semantics of pure DATALOG programs in the contex of mathematical logic.

We then informally specify an operational semantics for minimal DATALOG based on the *Recursive Query Subquery* algorithm. The algorithm was originally proposed by [Vieille 89] but we use the version found in [Ceri 90].

Further, we discuss in detail the subjects *unification* of the query and a corresponding rule, and the *selection function* which selects each literal from the rule body. The two subjects play a keyrole in our query answering algorithm.

Chapter 8

Conclusion

In this work we have extended the syntax of the deductive database language DATALOG and we have specified the well-formedness of this language. We have introduced methodes to optimize DATALOG programs and we have discussed how to determine the level of safety with the use of a data-flow analysis. We have specified a query answering algorithm for DATALOG.

In relation to this we have implemented a DATALOG syntax analyser, which also controls well-formedness integrity, and an interpreter based on the query answering algorithm.

8.1 Summary

Syntax

When specifying the syntax we paid much attension on the ease of using the language and the language's readability, i.e. how easy is it to understand the meaning of the database statements. By allowing arbitrary large expressions as arguments in predicates we succeded even better than PROLOG in our design of a userfriendly logic language. Further the syntax was extended to enable a manual specification of: 1) the adornment of predicates, 2) the domain of predicate arguments, and 3) the level of safety of predicates.

Optimizing

Our optimization strategy contains unfolding of non-recursive predicates, and optimization of redundant information, tautologies, and absurdities. The aim of the optimization is to cut down the irrelevant work of the interpreter.

The effect of the optimization depends on the program, e.g. there are programs that are optimal already, there are programs that performs orders of magnitudes better when optimized, and then there are programs that are not evaluable if not optimized.

Data-flow

The data-flow information on rules is constructed from the adornment of facts, standard predicates, and negated predicates. The analysis must ensure a safe use of the standard predicates and negated predicates, and when it does the program is said to be range restricted. This range restriction demand is critical to the semantcs of DATALOG and we can not quarantee a correct answer if this demand is violated.

Safety

When specifying databases it is important to know what level of safety can be guaranteed about each predicate in the database. Generaly we can state that if the database is range restricted and if the query answering algorithm terminates then the answer is correct. Further we can state that if the query-predicate does not depend on any arithmetics then the query answering algorithm is guaranteed to terminate.

Interpreter

When searching for a query anwering algorithm we decided to use an existing method, instead of developing a new one, and extend it with whatever facilities that was needed. Our choice of the QSQR algorithm was due to several reasons: 1) according to a benchmark the method performed orders of magnitudes better that other algorithms on non-linear predicates, 2) the implementation of the best performing bottom-up evaluation method was doubtful, and 3) our data-flow analysis gave us the opportunity of developing an intelligent selection function which was independent of the order of the literals in the rule body.

The reason why this is important is that it alow us to state that the language is declarative. At least it appears declarative to the user, i.e. no matter how the user permutates the literals the answer is the same because of our selection function.

Our selection function has a heuristics for choosing among several literals that are ready for selection. For each literal it computes a relative degree of instantiation. A literal with few possible solutions is harder instantiated than a literal with many possible solutions. The hardest instantiated literal is selected, because it will focus harder on the relevant solutions.

Implementaion

The DATALOG system we have implemented is an integrated system. The system consists of the interpreter, an editor and a program library all connected with a main menu. The system starts with an empty program. From the main menu it is then possible to load a new program from the library, interpret the actual program, to edit the program, or to save the program into the library. The system was implemented on a personal computer to run under DOS.

8.2 Future Work

Safety

We can not guarantee safety of predicates that uses arithmetics, a future work could extend the this definition of safety to include arithmetics. The problem is to state whether an arithmetical predicate is capable of inductively to introduce an infinite set of answers. We know that for a predicate to be unsafe it must be defined recursively and its arguments must be inductively incremented or decremented by an arithmetical predicate. This is however still a sufficient condition but it is not necessary.

Extensions

This system is easy to extend with other standard predicates, e.g. concatenation of strings, by defining the possible adornments and then to each adornment specify the semantics of the predicate.

Stratification is used to guarantee that there is exactly one fixed point for the evalutation of the program but there are non-stratified programs which also have just one fixed point and a possibility of evaluating such programs would be interesting. See section A.15 for such a program.

Optimization

There are two categories of optimization strategies: those which optimize with respect to the query and those which just optimize the database in general. We do only optimize with respect to the query by eliminating all rule that will not affect the query from the database. Future work could include partial evaluation of the program before the actual evaluation. Partial evaluation optimizes the program with respect to the query.

We consider unfolding of non-recursive predicates but one could also consider unfolding of base predicates, i.e. facts. But unfolding facts into rules are not an optimization if there are to many facts. So where is the limit of when to unfold facts? Well, we have two ideas of when to unfold facts: 1) if the predicate consists of one and only one fact, or 2) when the number of facts is comparable with the number of rules.

Selection function

Other heuristics would give other optimizations, what other heuristics and corresponding optimizations are possible should be investigated further.

Evaluation methods

Our interpreter could only handle finite solutions. We therefore introduced the range restriction demand. In future works one could consider a lazy evaluation of arithmetical and comparative predicates. This should make it possible to retur infinite solutions. The infinite solutions should be represented by the unsolved arithmetical and comparative predicates.

8.3 Applications

What is not possible to express is DATALOG is a comparison of the answer tuples itself, e.g. return only the largest solution. This leads to the idea of incorporating DATALOG in PROLOG. Another approach is to incorporate DATALOG under a graphical interface where rules are presented as heratical boxes as proposed in [JFN 91b].

If our interpreter could have handled large scale DATALOG programs, then we would have liked to experiment with developing meta-interpreters and study how to implement DATALOG programs, which itself could be interpreters, as data in a database as proposed in [JFN 91a].

If one, in a stepwise manner, extended the meta-interpreter to handle larger and larger data-programs, the procces of extending the meta-interpreter would perhaps convergate agains a general method and perhaps we would learn how automatically to construct DATALOG interpreters at any given size.

Chapter 9

Acknowledgements

Thanks to Leo Jensen Director of Prolog Development Center ApS for supplying me with a free user registration and a DOS version of their primary product [PDC PROLOG], in which I implemented the interpreter.

Prolog Development Center ApS is situated on H.J. Holst Vej 5A, DK-2605 Brøndby. Phone: +45 36 72 10 22, Fax: +45 36 72 02 69.

Bibliography

[Aho 86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Bancilhon 86a] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. *Magic sets and other strange ways to implement logic programs*, Proc. ACM-SIGMOD Symp. on Principles of Database Systems, Cambridge (MA), March 1986.

[Bancilhon 86b] François Bancilhon and Raghu Ramakrishnan. *An Amateur's Introduction to Recursive Query Processing Strategies*, In *Readings in Artificial Intelligence & Databases*, (eds. J. Mylopoulos & M.L. Brodie), Morgen Kaufmann Publ; 1989. First published in Proc. of the ACM-SIGMOD Conference, Washington D.C., May 1986.

[Bancilhon 89] François Bancilhon and Setrag Khoshafian. *A Calculus for Complex Objects.* Journal of Computer and System Sciences 38. pp. 326-340, 1989.

[Beeri 87] Catriel Beeri and Raghu Ramakrishnan *On the Power of Magic*, Symp. on Principles of Database Systems, 6, 1987. pp. 269-283.

[Bjørner 89] Dines Bjørner. *Software Architecture and Programming Systems Design, Volume I-V.* Department of Computer Science. Technical University of Denmark, 1989.

[Bondorf 90] Anders Bondorf. *Self-Applicable Partial Evaluation: Ph. D. Thesis*, Dept. of Computer Science, University of Copenhagen. DIKU Report No. 90/17. 1990. Chapter 7. Autoprojecting Prolog.

[Ceri 90] Stefano Ceri, Georg Gottlob, Letizia Tanca. *Logic Programming and Databases*, Surveys in Computer Science, Springer-Verlag, 1990.

[Deransart 85] Pierre Deransart and Jan Małuszyński. *Relating Logic Programs and Attribute grammars*, Journal of Logic Programming 1985:2:119-155.

[Fuller 88] David A. Fuller and Samson Abramsky. *Mixed Computation of Prolog Programs*, New Generation Computing 6. 1988. pp. 119-141.

[Jensen 84] Leo Jensen et al. *Prolog on a microcomputer*, Master's thesis, at the Department of Computer Science. Technical University of Denmark. 1984.

[Jones 86] Cliff B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice Hall, 1986.

[Kemp 88] David B. Kemp and Rodney W. Topor. *Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases*, Logic Programming. Proceedings of the Fifth International Conference and Symposium, 1988.

[Kemp 89] David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin and Krishnamurthy Meenakshi. *Propagating Constraints in Recursive Deductive Databases.* Logic Programming. Proceedings of the North American Conference, 1989.

[Kolaitis 90] Phokion G. Kolaitis and Moshe Y. Vardi. *On the Expressive Power of Datalog: Tools and a Case Study* pp. 61-71 Symp. on Principles of Database Systems, 9, 1990.

[Kowalski 79] Robert Kowalski. *Algorithm = Logic + Control*, Comm. of the ACM, July 1979, Volume 22, Number 7.

[Mumick 90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh and Raghu Ramakrishnan. *Magic Conditions* pp. 314-330. Symp. on Principles of Database Systems, 9, 1990.

[Nilsson 83] Jørgen Fischer Nilsson. *On The Compilation of a Domain-Based Prolog*, Technical report, Department of Computer Science. Technical University of Denmark. 1983.

[JFN 89] Jørgen Fischer Nilsson. *An Introduction to Logic Programming*, Lecture Notes, Department of Computer Science. Technical University of Denmark. ID Doc. no. 1033, January 1989.

[Nilsson 90] Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog*, John Wiley & Sons Ltd. 1990.

[JFN 90] Jørgen Fischer Nilsson. *Relational Knowledge Base Systems*, Technical report, Department of Computer Science. Technical University of Denmark. November 1991.

[JFN 91a] Jørgen Fischer Nilsson. *Meta-Datalog: Interpreting Datalog in Datalog*, Technical report, Department of Computer Science. Technical University of Denmark. March 1991.

[JFN 91b] Jørgen Fischer Nilsson. *TOPOLOG: Topological Visualization of Logical Knowledge Bases*, Technical report, Department of Computer Science. Technical University of Denmark. May 1991.

[PDC PROLOG] PDC PROLOG. DOS Version 3.21. Reference Guide, User's Guide and Toolbox. 1991. Prolog Development Center, H.J. Holst Vej 5A, DK-Brønby.

[Sacca 87] D. Sacca and C. Zaniolo. *Magic Counting Methods*, Proc. of the ACM-SIGMOD Conference, S. Francisco, May 1987.

[Sagiv 87] Yehoshua Sagiv *Optimizing Datalog Programs*, Symp. on Principles of Database Systems, 6, 1987. pp. 349-362.

[Stanat 77] Donald F. Stanat and David F. McAllister. *Discrete Mathematics in Computer Science*, Prentice Hall, 1977.

[Sterling 89] Leon Sterling and Randall D. Beer. *Metainterpreters for Expert System Construction*, The Journal of Logic Programming, 1989. pp. 163-178.

[Tamaki 84] Hisao Tamaki and Taisuke Sato. *Unfold/Fold Transformation of Logic Programs*, Proceedings of the Second International Logic Programming Conference. Uppsala, Sweden July 2-6, 1984.

[Ullman 89a] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*, Vol. 1+2, Computer Science Press, 1989.

[Ullman 89b] Jeffrey D. Ullman. *Bottom-Up Beats Top-Down for Datalog*, Symp. on Principles of Database Systems, 8, 1989. pp. 140-149.

[Van Emden 76] M. H. Van Emden and R. A. Kowalski. *The Semantics of Predicate Logic as a Programming Language*, Journal of the Association for Computing Machinery, Vol. 23, No. 4, October 1976, pp. 733-742.

[Vieille 89] Laurent Vieille. *Recursive Query Processing: The Power of Logic*, Theoretical Computer Science, Vol. 69, Fundamental Studies, 1989.

Appendix A

Examples of Datalog Programs

Each of the following sections contains either a program with the solution to the query or just an intensional database. All examples are annotated. If the interpreter could handle the program, then the result is presented. Some of the programs have already been seen before, but are repeated here to make the collection complete.

Unfortunately most larger programs could not be handled by the interpreter, because of the limitations of DOS, i.e. the 640 kbyte limit of main storage.

A.1 Railway Example

This example was found in [Ceri 90, page 217]. Consider a railroad network represented by the base predicate *link(X, Y)* stating that there is a direct link between station X and station Y with no intermediate station. On the basis of the base predicate *link*, we define several virtual predicates with the following associated intuitive meanings:

- *cutpoint(X,A,B)*: each connection (path) from station A to station B goes through station X.
- *connected(A,B)*: there is a connection between A and B involving one or more links.
- *circumvent(X,A,B)*: there is a connection between station A and station B which does not pass through station X.
- *existscutpoint(A,B)*: there is at least one cutpoint on the way from A to B.
- *safely_connected(A,B)*: A and B are connected, but there is no cutpoint for A and B. This means that if a station or link between A and B is disabled (e.g. by an accident), then there is still another way of reaching B from A.

- *linked(A,B)*: we assume that our network is an undirected graph, i.e., if A is linked to B then B is linked to A. However, the base predicate *link* is not necessarily symmetric since for expressing the fact that there is a link between A and B, it is sufficient to state either *link(A,B)* or *link(B,A)* but not necessarily both. The virtual predicate *linked* consists of the symmetric closure of the *link* predicate.
- *station*: X is a railway station.

Railway program.

```
% Database query %

   ?- existcutpoint(c,A).

   cutpoint{fb,fb,fb}.
   cutpoint(X,A,B)
      :- connected(A,B) &
         station(X) &
         not(circumvent(X,A,B)).


   circumvent{fb,fb,fb}.
   circumvent(X,A,B)
      :- not(eq(X,A)) &
         not(eq(X,B)) &
         linked(A,B) & station(X).
   circumvent(X,A,B)
      :- circumvent(X,A,C) &
         circumvent(X,C,B).

   linked{fb,fb}.
   linked(A,B) :- link(A,B).
   linked(A,B) :- link(B,A).


   connected{fb,fb}.
   connected(A,B) :- linked(A,B).
   connected(A,B)
      :- connected(A,C) &
         linked(C,B).

   existcutpoint{fb,fb}.
   existcutpoint(A,B)
      :- station(X) &
```

```
         cutpoint(X,A,B).



   safely_connected{fb,fb}.
   safely_connected(A,B)
      :- not(existcutpoint(A,B)) &
         connected(A,B).


% Equality %

   eq{fb,bb}.
   eq{bb,fb}.
   eq(X,X).

% The EDB of railway stations. %

   station{fb}.
   station(b).
   station(c).
   station(d).

% The EDB of links. %

   link{fb,fb}.
   link(b,c).
   link(c,d).
   link(d,b).
```

A.2 Fibonacci Function

The Fibonacci function maps natural numbers into the corresponding Fibonacci numbers. This example computes that `fib(12,233)` is true.

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.
```

```
-- SOURCE TEXT -----------------------
% Datalog specification of the Fibonacci function.%

?- fib(12,FIB).                  % The Query.                 %

fib(X,Y1+Y2) :- fib(X-1,Y1) &    % Recursive relation.        %
                fib(X-2,Y2).
fib(0,1).                        % Fact.                      %
fib(1,1).                        % Fact.                      %

fib[1] = [0;20].                 % Domain restriction.        %
fib:2.                           % Level of safety.           %
fib{fb,fb}.                      % Data-flow specification.   %

% Expected answer to the query:
N is the natural number and FIB is the
corresponding Fibonacci number.

N=0, FIB=1.
N=1, FIB=1.
N=2, FIB=2.
N=3, FIB=3.
N=4, FIB=5.
N=5, FIB=8.
N=6, FIB=13.
N=7, FIB=21.
N=8, FIB=34.
N=9, FIB=55.
N=10, FIB=89.
N=11, FIB=144.
N=12, FIB=233.
N=13, FIB=377.
N=14, FIB=610.
N=15, FIB=987.
N=16, FIB=1597.
N=17, FIB=2584.
N=18, FIB=4181.
N=19, FIB=6765.
N=20, FIB=10946.


%

-- OPTIMIZED INTENSIONAL DATABASE -----
```

```
REPEAT:
fib(X,V3)
   :-   V1 + 2 = X &
        fib(V1,Y2) &
        fib(V2,Y1) &
        V2 + 1 = X &
        Y1 + Y2 = V3.


-- EXTENSIONAL DATABASE ---------------
fib(0,1).
fib(1,1).

-- THE ANSWER TO THE QUERY ------------

FIB=233.
One solution.
```

A.3 Fibonacci Function

The Fibonacci function maps natural numbers into the corresponding Fibonacci numbers but we can also use it as the invers function.

In this example we find that to the Fibonacci number 8 corresponds the natural number 5.

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------
% Datalog specification of the Fibonacci function.%

?- fib(N,8).                        % The Query.                %

fib(X,Y1+Y2) :- fib(X-1,Y1) &       % Recursive relation.       %
                fib(X-2,Y2).
fib(0,1).                           % Fact.                     %
fib(1,1).                        % Fact.                       %

fib[1] = [0;14].                 % Domain restriction.         %
fib:2.                           % Level of safety.            %
fib{fb,fb}.                      % Data-flow specification.    %

% Expected answer to the query:

X=0, Y=1.
X=1, Y=1.
X=2, Y=2.
X=3, Y=3.
X=4, Y=5.
X=5, Y=8.
X=6, Y=13.
X=7, Y=21.
X=8, Y=34.
X=9, Y=55.
X=10, Y=89.
X=11, Y=144.
X=12, Y=233.
X=13, Y=377.
X=14, Y=610.
X=15, Y=987.
X=16, Y=1597.
X=17, Y=2584.
X=18, Y=4181.
X=19, Y=6765.
X=20, Y=10946.

11 solutions.

%

-- OPTIMIZED INTENSIONAL DATABASE -----
REPEAT:
fib(X,V3)
   :-   V1 + 2 = X &
fib(V1,Y2) &
fib(V2,Y1) &
V2 + 1 = X &
Y1 + Y2 = V3.


-- EXTENSIONAL DATABASE ---------------
```

```
fib(0,1).
fib(1,1).

-- THE ANSWER TO THE QUERY ------------

N=5.
One solution.
```

A.4 Albino

White wales and white bisons are well-known examples on albinos in the nature. Both parents must be albinos for the child to be an albino. That's why they are rare!

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------

?- albino(Albino).

albino(Child) :- parent(Child,Father) & albino(Father) &
                 parent(Child,Mother) & albino(Mother) &
                 Father != Mother.
albino(Child) :- albino_ext(Child).

parent(a,af).
parent(a,am).
parent(c,bf).
parent(c,cm).

albino_ext(af).
albino_ext(am).
albino_ext(bf).


albino      {fb}.
parent      {fb,fb}.
```

```
albino_ext {fb}.


% Expected answer to the query:

Albino=a.
Albino=af.
Albino=am.
Albino=bf.
4 solutions.

%
-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
albino(Child)
   :-   albino_ext(Child).

REPEAT:
albino(Child)
   :-   Father != Mother &
        parent(Child,Mother) &
        parent(Child,Father) &
        albino(Father) &
        albino(Mother).


-- EXTENSIONAL DATABASE ---------------
parent("a","af").
parent("a","am").
parent("c","bf").
parent("c","cm").
albino_ext("af").
albino_ext("am").
albino_ext("bf").

-- THE ANSWER TO THE QUERY ------------

Albino="a".
Albino="af".
Albino="am".
Albino="bf".
4 solutions.
```

A.5 Natural logarithm of two

A reduced version of the *ln2* example is presented.

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------
%Natural logarithm of two.%

 ?- ln2(X).

aux_ln2(X,0) :- X > 8.
aux_ln2(X,1/X-1/(X+1)+Y) :- aux_ln2(X+2,Y).
aux_ln2{bb,fb}.
aux_ln2[1] = [1;14].

ln2(X) :- aux_ln2(1,X).
ln2{fb}.


% Expected answer to the query:
  Several X' converging to ln(2).

X=0.69314718056
One solution

%

-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
aux_ln2(X,0)
   :-   8 < X.

REPEAT:
aux_ln2(X,V2)
   :-   X + 2 = V1 &
        aux_ln2(V1,Y) &
        V3 + V5 = V4 &
        V5 * V6 = 1 &
        X + 1 = V6 &
        V4 * X = 1 &
        V3 + Y = V2.

STEP:
ln2(X)
   :-   aux_ln2(1,X).


-- EXTENSIONAL DATABASE ---------------

-- THE ANSWER TO THE QUERY ------------

X=533/840.         = 0.63452
X=1627/2520.       = 0.64563
X=18107/27720.     = 0.65321
3 solutions.
```

A.6 Small meta-interpret

Read more about interpreting datalog in datalog in [JFN 91a]. Unfortunately the program was too large to run on this version of the interpreter.

```
% Meta-Datalog, Interpreting Datalog in Datalog. %

%?- e1(P).%

e1(P)        :- b3(P,if,Q) & e1(Q).
e2(P,X1)     :- b3(P,if,Q) & e2(Q,X1).

e1(P)        :- b4(P,and,Q1,Q2) & e1(Q1) & e1(Q2).
e2(P,X1)     :- b4(P,and,Q1,Q2) & e2(Q1,X1) & e2(Q2,X1).

e1(P)        :- b3(P,exist,Q) & e2(Q,_).
e2(P,X1)     :- b3(P,exist,Q) & e3(Q,X1,_).

e2(P,_)      :- b3(P,exp,Q) & e1(Q).
e3(P,_,X1)   :- b3(P,exp,Q) & e2(Q,X1).

e2(P,X1)     :- b3(P,ref,Q) & e3(Q,X1,X1).

e3(P,X1,X2) :- b3(P,inv,Q) & e3(Q,X2,X1).
```

```
e1(true).
e3(id,X,X).

e1(P)        :- b1(P).
e2(P,X1)     :- b2(P,X1).
e3(P,X1,X2)  :- b3(P,X1,X2).


b1{fb}.
b2{fb,fb}.
b3{fb,fb,fb}.
b4{fb,fb,fb,fb}.

e1{fb}.
e2{fb,ff}.
e3{fb,ff,ff}.
```

A.7 Larger meta-interpreter

This example is also from [JFN 91a]. The larger the meta-datalog interpreter is, the larger is the subset of DATALOG which it can interpret.

```
% Meta-Datalog, Interpreting Datalog in Datalog. %

%?- e2(P,X).%

e1(P)               :- b3(P,if,Q) & e1(Q).
e2(P,X1)            :- b3(P,if,Q) & e2(Q,X1).
e3(P,X1,X2)         :- b3(P,if,Q) & e3(Q,X1,X2).
e4(P,X1,X2,X3)      :- b3(P,if,Q) & e4(Q,X1,X2,X3).

e1(P)               :- b4(P,and,Q1,Q2) & e1(Q1) & e1(Q2).
e2(P,X1)            :- b4(P,and,Q1,Q2) & e2(Q1,X1) & e2(Q2,X1).
e3(P,X1,X2)         :- b4(P,and,Q1,Q2) &
                       e3(Q1,X1,X2) & e3(Q2,X1,X2).
e4(P,X1,X2,X3)      :- b4(P,and,Q1,Q2) &
                       e4(Q1,X1,X2,X3) & e4(Q2,X1,X2,X3).

e1(P)               :- b3(P,exist,Q) & e2(Q,_).
e2(P,X1)            :- b3(P,exist,Q) & e3(Q,X1,_).
e3(P,X1,X2)         :- b3(P,exist,Q) & e4(Q,X1,X2,_).
e4(P,X1,X2,X3)      :- b3(P,exist,Q) & e5(Q,X1,X2,X3,_).

e2(P,_)             :- b3(P,exp,Q) & e1(Q).
```

```
e3(P,_,X1)          :- b3(P,exp,Q) & e2(Q,X1).
e4(P,_,X1,X2)       :- b3(P,exp,Q) & e3(Q,X1,X2).
e5(P,_,X1,X2,X3)    :- b3(P,exp,Q) & e4(Q,X1,X2,X3).

e2(P,X1)            :- b3(P,ref,Q) & e3(Q,X1,X1).
e3(P,X1,X2)         :- b3(P,ref,Q) & e4(Q,X1,X1,X2).
e4(P,X1,X2,X3)      :- b3(P,ref,Q) & e5(Q,X1,X1,X2,X3).

e3(P,X1,X2)         :- b3(P,inv,Q) & e3(Q,X2,X1).
e4(P,X1,X2,X3)      :- b3(P,inv,Q) & e4(Q,X2,X1,X3).
e5(P,X1,X2,X3,X4) :- b3(P,inv,Q) & e5(Q,X2,X1,X3,X4).

e4(P,X1,X2,X3)      :- b3(P,rot,Q) & e4(Q,X2,X3,X1).
e5(P,X1,X2,X3,X4) :- b3(P,rot,Q) & e5(Q,X2,X3,X4,X1).


e1(true).
e3(id,X,X).

e1(P)               :- b1(P).
e2(P,X1)            :- b2(P,X1).
e3(P,X1,X2)         :- b3(P,X1,X2).
e4(P,X1,X2,X3)      :- b4(P,X1,X2,X3).
e5(P,X1,X2,X3,X4) :- b5(P,X1,X2,X3,X4).
```

A.8 Illegal Datalog Program

This example illustrates what happens when the user takes responsibility for range restriction in a program that will violate this demand. See also section 4.4.2.

```
% This should cause a run-time error
  or a break-down of the system. %

  ?- q(Y).

  q(Y) :- p(X) & Y=X/X & not(Y != X).

  p(2).
  p(1).
  p(0).

  p{fb}.
  q{fb}.
  q:3. % User responsible safety. %

% Expected answer to the query:

RUN-TIME ERROR: Illegal use of standard predicate.

%
```

A.9 Zero divided by zero

The statement `Y * 0 = 0` is true for all values of `Y`, i.e. the expression `Y= 0 / 0` will not bind `Y` to a value but to an anonymous variable.

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------

  ?- q(Y).

  q(Y) :- p(X) & Y=X/X.

  p(2).
  p(1).
  p(0).

  p{fb}.
  q{fb}.

  q:3. % User responsible safety. %


% Expected answer to the query:

Y=1.
Y=_.
2 solutions.

%
```

```
-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
q(V1)
   :-   p(X) &
        V1 * X = X.


-- EXTENSIONAL DATABASE ---------------
p(2).
p(1).
p(0).

-- THE ANSWER TO THE QUERY ------------

Y=1.
Y=_.
2 solutions.
```

A.10 Linear Ancestor Example

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------
% Linear ancestor program.%

?- ancestor(aa,X).

ancestor(X,Y) :- parent(X,Z) & ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).

parent(a,aa).
parent(a,ab).
parent(aa,aaa).
parent(aa,aab).
parent(aaa,aaaa).
parent(c,ca).

% Expected answer to the query:

X=aaa.
X=aab.
X=aaaa.
3 solutions.

%

-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
ancestor(X,Y)
   :-   parent(X,Y).

REPEAT:
ancestor(X,Y)
   :-   ancestor(Z,Y) &
        parent(X,Z).


-- EXTENSIONAL DATABASE ---------------
parent("a","aa").
parent("a","ab").
parent("aa","aaa").
parent("aa","aab").
parent("aaa","aaaa").
parent("c","ca").

-- THE ANSWER TO THE QUERY ------------

X="aaa".
X="aaaa".
X="aab".
3 solutions.
```

A.11 Non Linear Ancestor Example

```
-- MESSAGES ---------------------------
```

```
The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT -----------------------
% Linear ancestor program.%

?- ancestor(aa,X).

ancestor(X,Y) :- ancestor(X,Z) & ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).

parent(a,aa).
parent(a,ab).
parent(aa,aaa).
parent(aa,aab).
parent(aaa,aaaa).
parent(c,ca).

-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
ancestor(X,Y)
   :-   parent(X,Y).

REPEAT:
ancestor(X,Y)
   :-   ancestor(Z,Y) &
        ancestor(X,Z).


-- EXTENSIONAL DATABASE ---------------
parent("a","aa").
parent("a","ab").
parent("aa","aaa").
parent("aa","aab").
parent("aaa","aaaa").
parent("c","ca").

-- THE ANSWER TO THE QUERY ------------

X="aaa".
X="aaaa".
```

```
X="aab".
3 solutions.
```

A.12 Same Generation Example

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------
% Linear same generation program. %

?- sg(aa,X).

sg(XP,YP) :- parent(X,XP) & parent(Y,YP) & sg(X,Y).
sg(X,X).

parent(a,aa).
parent(a,ab).
parent(aa,aaa).
parent(aa,aab).
parent(aaa,aaaa).
parent(c,ca).

% Expected answer to the query:

X=aa.
X=ab.
2 solutions.

%

-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
sg(X,X).
REPEAT:
sg(XP,YP)
   :-   sg(X,Y) &
```

```
        parent(X,XP) &
        parent(Y,YP).


-- EXTENSIONAL DATABASE ---------------
parent("a","aa").
parent("a","ab").
parent("aa","aaa").
parent("aa","aab").
parent("aaa","aaaa").
parent("c","ca").

-- THE ANSWER TO THE QUERY ------------

X="aa".
X="ab".
2 solutions.
```

A.13 Same Generation Example

John is of same generation as George, because Mary is their common grandmother.

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------

?- sg(john,X).

sg(X,Y) :- sgc(X,Y) & not(X=Y).

sgc(X,X).
sgc(X,Y) :- par(X,X1) & sgc(X1,Y1) & par(Y,Y1).

par(john,henry).
par(george,jim).
par(henry,mary).
par(jim,mary).

% Expected answer to the query:

X=george.
One solution.

%

-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
sgc(X,X).
REPEAT:
sgc(X,Y)
   :-   par(Y,Y1) &
        par(X,X1) &
        sgc(X1,Y1).

STEP:
sg(X,Y)
   :-   X != Y &
        sgc(X,Y).


-- EXTENSIONAL DATABASE ---------------
par("john","henry").
par("george","jim").
par("henry","mary").
par("jim","mary").

-- THE ANSWER TO THE QUERY ------------

X="george".
One solution.
```

A.14 The Alpine Club

In the Alpine Club all members are skiers or (mountain) climbers. All climbers do not like rain. All skiers like snow. Tony, Mike, and John are members of the

Alpine Club. Mike likes what Tony dislikes and Mike dislikes what Tony likes. Tony likes both rain and snow. Who among the members are climbers?

```
-- MESSAGES --------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT -----------------------
%Alpine Club%

?-        climber(X).

member    (tony).
member    (mike).
member    (john).

climber   (X)      :-  member(X) & dislike(X,rain).
skier     (X)      :-  member(X) & like(X,snow).

like      (tony,rain).
like      (tony,snow).
like      (mike,X) :-   dislike(tony,X).
dislike   (mike,X) :-   like(tony,X).


-- OPTIMIZED INTENSIONAL DATABASE -----
REPEAT:
like("mike",X)
   :-   dislike("tony",X).

dislike("mike",X)
   :-   like("tony",X).

STEP:
climber(X)
   :-   dislike(X,"rain") &
        member(X).


-- EXTENSIONAL DATABASE ---------------
member("tony").
member("mike").
member("john").
like("tony","rain").
like("tony","snow").

-- THE ANSWER TO THE QUERY ------------

X="mike".
One solution.
```

A.15 Stratification check

This example shows that the stratification check works. The following rule is not stratified.

```
like      (mike,X) :-   weather(X) & not(like(tony,X)).
```

This is an example of a program that is not stratified, but does have a unique fixed point answer.

Find the `FATAL ERROR` message below.

```
-- MESSAGES ---------------------------

FATAL ERROR: The database is not stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------
%Alpine Club%

?-        skier(X).

member    (tony).
member    (mike).
member    (john).

climber   (X)      :-   member(X) & not(like(X,rain)).
skier     (X)      :-   member(X) & like(X,snow).

like      (tony,rain).
like      (tony,snow).
like      (mike,X) :-   weather(X) & not(like(tony,X)).

weather(snow).
weather(rain).


-- OPTIMIZED INTENSIONAL DATABASE -----

-- EXTENSIONAL DATABASE ---------------

-- THE ANSWER TO THE QUERY ------------
NO
```

A.16 Arity check

This example shows that the arity check works. Find the `FATAL ERROR` message below.

```
-- MESSAGES ---------------------------

The database is stratified.
FATAL ERROR: The following predicates conflict in arity: p.
FATAL ERROR: Arity of query conflicts with the database.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------

?- p(X,Y).

p(1).
p(2,3).
p(4,5,6).
p(7,8,9,10).

-- OPTIMIZED INTENSIONAL DATABASE -----

-- EXTENSIONAL DATABASE ---------------

-- THE ANSWER TO THE QUERY ------------
NO
```

A.17 Fail check

This example shows that the fail check works. Find the `FATAL ERROR` message below.

```
-- MESSAGES --------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
FATAL ERROR: Standard predicate fail defined by the user.


-- SOURCE TEXT -----------------------

?- fail.

fail.
-- OPTIMIZED INTENSIONAL DATABASE -----

-- EXTENSIONAL DATABASE ---------------

-- THE ANSWER TO THE QUERY ------------
NO
```

A.18 Boolean Logic

Implementaion of boolean logic in DATALOG.

```
-- MESSAGES --------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT -----------------------
```

```
% boolsk regning. %
?- query(X,Y,Z).
query(X,Y,Z) :- and(X,Y,Z) & non(X,Y).

xor(X,X,false) :- domain(X).
xor(X,Y,true)  :- non(X,Y).

non(X,Y) :- not(equal(Y,X)) & domain(X) & domain(Y).

and(true,X,X)      :- domain(X).
and(false,X,false) :- domain(X).
or(true,X,true)    :- domain(X).
or(false,X,X)      :- domain(X).

equal(X,X) :- domain(X).

domain(true).
domain(false).


-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
equal(X,X)
   :-   domain(X).

STEP:
and("true",X,X)
   :-   domain(X).

and("false",X,"false")
   :-   domain(X).

STEP:
non(X,Y)
   :-   domain(Y) &
        not(equal(Y,X)) &
        domain(X).

STEP:
query(X,Y,Z)
   :-   non(X,Y) &
        and(X,Y,Z).
```

```
-- EXTENSIONAL DATABASE ---------------
domain("true").
domain("false").

-- THE ANSWER TO THE QUERY ------------

X="false", Y="true", Z="false".
X="true", Y="false", Z="false".
2 solutions.
```

A.19 Transformation of negated expression

Only the = sign is negated, the rest is transformed into positive expressions.

```
-- MESSAGES ---------------------------

The database is stratified.
Arity check done.
Range check done.
WARNING: No Type check done.
Fail undefined.


-- SOURCE TEXT ------------------------
% Example of expression rewriting on a negated expression.  %

?- p5.

p5 :- p(5).
p(X) :- not(X=2-1/2+3+5/10) & X<100.
p{bb}.

% Expected answer to the query:
NO

%
-- OPTIMIZED INTENSIONAL DATABASE -----
STEP:
p(X)
   :-   X < 100 &
X != V1 &
V3 + 3 = V2 &
V4 * 2 = 1 &
```

```
V3 + V4 = 2 &
V5 * 10 = 5 &
V2 + V5 = V1.

STEP:
p5
   :-   p(5).


-- EXTENSIONAL DATABASE ---------------

-- THE ANSWER TO THE QUERY ------------
NO
```

Appendix B

Well-formedness

B.1 Stratification of the Database

Abstract Domains

```
1.     $STRATUM = ID \xrightarrow[m]{} \mathsf{N}_1$

2.     $stratified(db) \triangleq$
 .1      let $stratum = [\ p \mapsto 1 \mid$ mk-$RULE($mk-$HEAD(p,),) \in db\ ]$ in
 .2        let $NoPredicates =$ card dom $stratum$ in
 .3          $strata(db, stratum, [\,], NoPredicates) \leq NoPredicates$
2.4    type: $stratified : DATABASE \rightarrow BOOL$
```

Annotations to *stratified*:

.1 Start by setting all predicates' stratum to 1.
.2 Number of predicates in the program.
.3 If the largest stratum is larger than the number of predicates then the program is not stratified.

End of annotations

```
3.     $strata(db, stratum_1, stratum_2, np) \triangleq$
 .1      $max($ rng $stratum_1) > np \rightarrow max($ rng $stratum_1),$
 .2      $stratum_1 = stratum_2 \qquad \rightarrow max($rng $stratum_1),$
 .3      $\mathsf{T} \qquad\qquad\qquad\qquad \rightarrow$
 .4        ( let $stratum_3 = [\ p \mapsto local(r, stratum_1) \mid$
 .5              $r \in db : r =$ mk-$RULE($mk-$HEAD(p,),)\ ]$ in
 .6            $strata(rs, stratum_3, stratum_1, np)$ )
3.7    type: $strata :\ DATABASE \times STRATUM$
 .8                $\times\ STRATUM \times \mathsf{INTG} \rightarrow \mathsf{INTG}$
```

Annotations to *strata*:

.1 Stop the recursion if either the largest stratum is larger than the number of predicates or

.2 if the latest computation did not change any stratum.

.3 − .6 Otherwise continue the iteration.

End of annotations

```
4.   local(mk-RULE((p,),body),stratum) ≜
 .1     let pos = { stratum(q) | mk-POS(mk-HEAD(q,)) ∈ body },
 .2         neg = { stratum(q) + 1 | mk-NEG(mk-HEAD(q,)) ∈ body } in
 .3       max(union(pos,neg,{ stratum(p) }))
4.4  type: local : RULE × STRATUM → INTG
```

Annotations to *local*:

.1 − .2 *Comparative predicates* do not affect the computation. Collect the strata from the *body*.

.3 In order to stratify the database, each predicate must have at least the same stratum as the positive atoms and at least a stratum that is one larger than the negative atoms.

End of annotations

B.2 Aritycheck

Abstract Domains

```
1.   MAP = ID $\xrightarrow[m]{}$ N_0

2.   aritycheck(mk-HEAD(q,ts),db) ≜
 .1     let map = aritydb(db,[ ]) in
 .2        { i | p ∈ dom map, i = map(p): i = CONFLICT } = { } ∧
 .3        (q ∈ dom map ⟹ map(q) = len ts)
2.4  type: aritycheck : DATABASE → BOOL
```

Annotations to *aritycheck*:

.1 Construct a table from predicate symbols into the corresponding arity or perhaps a CONFLICT.

.2 Are there any arity-conflicts in the database?

.3 Aritycheck of the query.

End of annotations

```
3.     $aritydb(db,map) \triangleq$
 .1       $db = \{\ \} \rightarrow map,$
 .2       $\top \qquad \rightarrow$
 .3          let $r \in db$ in
 .4          let $map_1 = aritydb(db \setminus \{r\},map)$ in
 .5             $arityrule(r,map_1)$
3.6    type: $aritydb : DATABASE \times MAP \rightarrow MAP$
```

Annotations to *aritydb*:

.1 − .5 Check all rules in the database one at a time.

End of annotations

```
4.     $arityrule($mk-$RULE(head,body),map) \triangleq$
 .1       $aritybody(\{$ mk-$POS(HEAD)\} \cup body,map)$
4.2    type: $arityrule : RULE \times MAP \rightarrow MAP$
```

Annotations to *arityrule*:

.1 Check the head and the body together

End of annotations

```
5.     $aritybody(body,map) \triangleq$
 .1       $body = \{\ \} \rightarrow map,$
 .2       $\top \qquad \rightarrow$
 .3          let $l \in body$ in
 .4          let $map_1 = aritybody(db \setminus \{l\},map)$ in
 .5          cases $l$ :
 .6             mk-$POS(atom) \rightarrow arityatom(atom,map_1),$
 .7             mk-$NEG(atom) \rightarrow arityatom(atom,map_1)$
5.8    type: $aritybody : BODY \times MAP \rightarrow MAP$
```

Annotations to *aritybody*:

.1 − .4 Check one literal at a time.

.5 − .7 Their is no difference between positive and negative literals.

End of annotations

```
6.    arityatom(atom,map) ≜
 .1     cases atom :
 .2       mk-HEAD(p,ts) →
 .3         p ∈ dom map →
 .4           if map(p) = len ts
 .5           then map
 .6           else  map + [ p ↦ CONFLICT ],
 .7         T              → map ∪ [ p ↦ len ts ]
 .8       T                → map,
6.9   type: arityatom : ATOM × MAP → MAP
```

Annotations to *arityatom*:

.2−.7 Count the arity, in case the predicate has occurred before comparing the information, otherwise just add the information.

.8 Comparative predicates do not contribute with any information.

End of annotations

B.3 Standard predicate may not be defined by the user

```
7.    failcheck(db) ≜
 .1     (∀mk-RULE(mk-HEAD(p,),) ∈ db)(p ≠ fail)
7.2   type: failcheck : DATABASE → BOOL
```

Annotations to *failcheck*:

.1 For all rules in the database, the predicate name is not fail.

End of annotations

Appendix C

Optimizing Extended Datalog Programs

C.1 Transformation Function

```
1.    $\mathcal{T}$-DATABASE(db) $\triangleq$
 .1      $db = \{\ \} \rightarrow \{\ \},$
 .2      $\top \qquad \rightarrow$
 .3         let $r \in db$ in
 .4            $\{\ \mathcal{T}$-RULE $(r)\} \cup \mathcal{T}$-DATABASE$(db \setminus \{r\})$
1.5   type: $\mathcal{T}$-DATABASE : DATABASE $\rightarrow$ DATABASE
```

Annotations to $\mathcal{T}$-*DATABASE*:

.1−.4 The transformation is local to the *rules*, so they are treated one by one.

End of annotations

```
2.    $\mathcal{T}$-RULE(r) $\triangleq$
 .1      let mk-RULE (head,body) = r in
 .2      let varset = all-var-rule(r) in
 .3      let $(body_1,varset_1) = \mathcal{T}$-BODY(body,varset) in
 .4      let $(body_2,head_1,) = \mathcal{T}$-HEAD$(head,varset_1)$ in
 .5         mk-RULE $(head_1,body_1 \cup body_2)$
2.6   type: $\mathcal{T}$-RULE : RULE $\rightarrow$ RULE
```

Annotations to $\mathcal{T}$-*RULE*:

.2 *varset* contains the variable names in the rule.

.3 Transform the *body*

.4 Transform the *head*

.5 The function returns the transformed *rule*

End of annotations

```
3.    $\mathcal{T}$-BODY(body,varset) $\triangleq$
 .1     body = { } → ({ }, varset),
 .2     T          →
 .3       let $l \in$ body in
 .4       let (body$_1$,varset$_1$) = $\mathcal{T}$-LITERAL(l,varset) in
 .5       let (body$_2$,varset$_2$) = $\mathcal{T}$-BODY(body \ { l },varset$_1$) in
 .6         (body$_1$ $\cup$ body$_2$ ,varset$_2$)
3.7   type: $\mathcal{T}$-BODY : BODY $\times$ VARSET → BODY $\times$ VARSET
```

Annotations to $\mathcal{T}$-*BODY*:

.1−.6 Transform the *literals* one by one.

End of annotations

```
4.    $\mathcal{T}$-LITERAL(l,varset) $\triangleq$
 .1     cases l :
 .2       mk-POS(atom) →
 .3         let (body,atom$_1$,varset$_1$) = $\mathcal{T}$-ATOM(atom,varset) in
 .4           (body $\cup$ {mk-POS(atom$_1$)},varset$_1$),
 .5       mk-NEG(atom)→
 .6         let (body,atom$_1$,varset$_1$) = $\mathcal{T}$-ATOM(atom,varset) in
 .7           (body $\cup$ {mk-NEG(atom$_1$)},varset$_1$)
4.8   type: $\mathcal{T}$-LITERAL : LITERAL $\times$ VARSET → BODY $\times$ VARSET
```

Annotations to $\mathcal{T}$-*LITERAL*:

.1−.7 There are *positive* and *negative* literals.

End of annotations

```
5.    $\mathcal{T}$-ATOM(atom,varset) $\triangleq$
 .1     cases atom :
 .2       mk-HEAD() → $\mathcal{T}$-HEAD(atom,varset)
 .3       T         → $\mathcal{T}$-BUILT-IN(atom,varset)
5.4   type: $\mathcal{T}$-ATOM : ATOM $\times$ VARSET → BODY $\times$ ATOM $\times$ VARSET
```

Annotations to $\mathcal{T}$-*ATOM*:

.1−.8 Atoms are either *heads* or *built-in standard predicates.*

End of annotations

```
6.    $\mathcal{T}$-HEAD(head,varset) $\triangleq$
 .1     let mk-HEAD(p,ts) = head in
 .2       let (body,$ts_1$,$varset_1$) = $\mathcal{T}$-TERMLIST(ts,varset) in
 .3         (body,mk-HEAD(p,$ts_1$),$varset_1$)
6.4   type: $\mathcal{T}$-HEAD : HEAD $\times$ VARSET $\rightarrow$ BODY $\times$ HEAD $\times$ VARSET
```

Annotations to $\mathcal{T}$-*HEAD*:

.1 − .3 Transforms the *head*

End of annotations

```
7.    $\mathcal{T}$-TERMLIST(ts,varset) $\triangleq$
 .1     ts = < > $\rightarrow$ ({ },< >,varset),
 .2     T        $\rightarrow$
 .3       let ($body_1$,$ts_1$,$varset_1$) = $\mathcal{T}$-TERMLIST(tl ts,varset) in
 .4         let ($body_2$,v,$varset_2$) = $\mathcal{T}$-EXPR(hd ts,$varset_1$) in
 .5           ($body1_1$ $\cup$ $body_2$, v ^ $ts_1$,$varset_2$)
7.6   type: $\mathcal{T}$-TERMLIST :  TERMLIST $\times$ VARSET $\rightarrow$
 .7                                BODY $\times$ TERMLIST $\times$ VARSET
```

Annotations to $\mathcal{T}$-*TERMLIST*:

.1 − .10 Transform one *term* at a time.

End of annotations

```
8.     $\mathcal{T}$-BUILT-IN(atom,varset$_1$) $\triangleq$
 .1      cases opr :
 .2        mk-NOT-EQUAL(e$_1$,e$_2$)          $\rightarrow$
 .3          let (body$_1$,v$_1$,varset$_2$) = $\mathcal{T}$-EXPR(e$_1$,varset$_1$)in
 .4          let (body$_2$,v$_2$,varset$_3$) = $\mathcal{T}$-EXPR(e$_2$,varset$_2$)in
 .5          (body$_1$ $\cup$ body$_2$,mk-NOT-EQUAL(v$_1$,v$_2$),varset$_3$),
 .6        mk-EQUAL(e$_1$,e$_2$)              $\rightarrow$
 .7          let (body$_1$,v$_1$,varset$_2$) = $\mathcal{T}$-EXPR(e$_1$,varset$_1$)in
 .8          let (body$_2$,v$_2$,varset$_3$) = $\mathcal{T}$-EXPR(e$_2$,varset$_2$)in
 .9          (body$_1$ $\cup$ body$_2$,mk-EQUAL(v$_1$,v$_2$),varset$_3$),
 .10       mk-LESS-THAN(e$_1$,e$_2$)          $\rightarrow$
 .11         let (body$_1$,v$_1$,varset$_2$) = $\mathcal{T}$-EXPR(e$_1$,varset$_1$)in
 .12         let (body$_2$,v$_2$,varset$_3$) = $\mathcal{T}$-EXPR(e$_2$,varset$_2$)in
 .13         (body$_1$ $\cup$ body$_2$,mk-LESS-THAN(v$_1$,v$_2$),varset$_3$),
 .14       mk-LESS-EQUAL(e$_1$,e$_2$)         $\rightarrow$
 .15         let (body$_1$,v$_1$,varset$_2$) = $\mathcal{T}$-EXPR(e$_1$,varset$_1$)in
 .16         let (body$_2$,v$_2$,varset$_3$) = $\mathcal{T}$-EXPR(e$_2$,varset$_2$)in
 .17         (body$_1$ $\cup$ body$_2$,mk-LESS-EQUAL(v$_1$,v$_2$),varset$_3$),
 .18       mk-GREATER-THAN(e$_1$,e$_2$)   $\rightarrow$
 .19         let (body$_1$,v$_1$,varset$_2$) = $\mathcal{T}$-EXPR(e$_1$,varset$_1$)in
 .20         let (body$_2$,v$_2$,varset$_3$) = $\mathcal{T}$-EXPR(e$_2$,varset$_2$)in
 .21         (body$_1$ $\cup$ body$_2$,mk-LESS-THAN(v$_2$,v$_1$),varset$_3$),
 .22       mk-GREATER-EQUAL(e$_1$,e$_2$) $\rightarrow$
 .23         let (body$_1$,v$_1$,varset$_2$) = $\mathcal{T}$-EXPR(e$_1$,varset$_1$)in
 .24         let (body$_2$,v$_2$,varset$_3$) = $\mathcal{T}$-EXPR(e$_2$,varset$_2$)in
 .25         (body$_1$ $\cup$ body$_2$,mk-LESS-EQUAL(v$_2$,v$_1$),varset$_3$)
8.26   type: $\mathcal{T}$-BUILT-IN :  BUILT-IN $\times$ VARSET $\rightarrow$
 .27                           BODY $\times$ BUILT-IN $\times$ VARSET
```

Annotations to $\mathcal{T}$-*BUILT-IN*:

.1 – .26 The new *comparative predicate* is constructed from the operator and the two introduced variables. During this transformation GREATER THAN and GREATER THAN OR EQUAL is transformed into LESS THAN and LESS THAN OR EQUAL respectively.

End of annotations

```
9.    $\mathcal{T}$-$EXPR(expr,varset) \triangleq$
.1       **let** $t \in$ TOKEN **be s.t.** $t \notin varset$ **in**
.2       **let** $v =$ **mk-**$VAR(t)$ **in**
.3       **let** $varset_1 = varset \cup \{ v \}$ **in**
.4          **cases** $expr$ :
.5             **mk-**$PLUS(e_1,e_2) \quad \rightarrow$
.6                **let** $(body_1,v_1,varset_2) = \mathcal{T}$-$EXPR(e_1,varset_1)$ **in**
.7                **let** $(body_2,v_2,varset_3) = \mathcal{T}$-$EXPR(e_2,varset_2)$ **in**
.8                $(\{$ **mk-**$EQUAL($**mk-**$PLUS(v_1,v_2),v)\} \cup body_1 \cup body_2,$
.9                 $v,varset_3)$
.10            **mk-**$MINUS(e_1,e_2) \rightarrow$
.11               **let** $(body_1,v_1,varset_2) = \mathcal{T}$-$EXPR(e_1,varset_1)$ **in**
.12               **let** $(body_2,v_2,varset_3) = \mathcal{T}$-$EXPR(e_2,varset_2)$ **in**
.13               $(\{$ **mk-**$EQUAL($**mk-**$PLUS(v,v_2),v_1)\} \cup body_1 \cup body_2,$
.14                $v,varset_3)$
.15            **mk-**$MULT(e_1,e_2) \quad \rightarrow$
.16               **let** $(body_1,v_1,varset_2) = \mathcal{T}$-$EXPR(e_1,varset_1)$ **in**
.17               **let** $(body_2,v_2,varset_3) = \mathcal{T}$-$EXPR(e_2,varset_2)$ **in**
.18               $(\{$ **mk-**$EQUAL($**mk-**$MULT(v_1,v_2),v)\} \cup body_1 \cup body_2,$
.19                $v,varset_3)$
.20            **mk-**$DIV(e_1,e_2) \qquad \rightarrow$
.21               **let** $(body_1,v_1,varset_2) = \mathcal{T}$-$EXPR(e_1,varset_1)$ **in**
.22               **let** $(body_2,v_2,varset_3) = \mathcal{T}$-$EXPR(e_2,varset_2)$ **in**
.23               $(\{$ **mk-**$EQUAL($**mk-**$MULT(v,v_2),v_1)\} \cup body_1 \cup body_2,$
.24                $v,varset_3)$
.25            **mk-**$SIGN(e_1) \qquad \rightarrow$
.26               **let** $(body_1,v_1,varset_2) = \mathcal{T}$-$EXPR(e_1,varset_1)$ **in**
.27               $(\{$ **mk-**$EQUAL($**mk-**$SIGN(v_1),v)\} \cup body_1,$
.28                $v,varset_2)$
.29            T                      $\rightarrow (\{ \},expr,varset)$
9.30  **type**: $\mathcal{T}$-$EXPR$: $EXPR \times VARSET \rightarrow BODY \times VAR \times VARSET$
```

Annotations to $\mathcal{T}$-*EXPR*:

.1 – .3 Introduce a new variable and add it to *varset*

.4 – .28 Transform subexpressions, return the body, the introduced variable and the variable set.

.29 Simple expressions are not transformed and the introduced variable is not used.

End of annotations

C.2 Iterative optimization strategy

Not all optimization strategies have been specified. We have just specified enough to obtain the smaller language and to ensure a simplification of the interpreter. What have been specified have also been implemented in our system.

C.2.1 Transform negative comparative predicates

```
1.    negcon2pos(l) ≜
 .1     cases l :
 .2       mk-NEG(mk-EQUAL(e_1,e_2))          →
 .3         mk-POS(mk-NOT-EQUAL(e_1,e_2)),
 .4       mk-NEG(mk-LESS-THAN(e_1,e_2))      →
 .5         mk-POS(mk-LESS-EQUAL(e_2,e_1)),
 .6       mk-NEG(mk-LESS-EQUAL(e_1,e_2))     →
 .7         mk-POS(mk-LESS-THAN(e_2,e_1)),
 .8       mk-NEG(mk-NOT-EQUAL(e_1,e_2))      →
 .9         mk-POS(mk-EQUAL(e_1,e_2)),
 .10      T                                  →l
1.11  type: negcon2pos: LITERAL → LITERAL
```

Annotations to *negcon2pos*:

.2 not(X=Y) is translated into X != Y

.4 not(X<Y) is translated into Y <= X

.6 not(X<=Y) is translated into Y <X

.8 not(X!=Y) is translated into X = Y

End of annotations

C.2.2 Remove simple bindings

```
2.    optimize-equal(r_1) ≜
 .1     let (set,r_2) = find-eq-rule(r_1) in
 .2       𝒮-set(set,r2)
2.3   type: optimize-equal : RULE → RULE
```

Annotations to *optimize-equal*:

.1 Try to find one positive *equal* atom and

.2 Substitute it. One can only substitute and remove one equal atom at a time, the *optimize-equal* function has to be applied to the rule, maybe several times, until all equal atoms are removed.

End of annotations

Auxiliary Domains

```
1.      $SSET = BINDING\text{-}\underline{\textbf{set}}$

2.      $find\text{-}eq\text{-}rule(\underline{\textbf{mk-}}RULE(head,body)) \triangleq$
 .1        $\underline{\textbf{let}}$ $(set,body_2) = find\text{-}eq\text{-}body(body_1)$ $\underline{\textbf{in}}$
 .2           $(set,\underline{\textbf{mk-}}RULE(head,body_2))$
2.3     $\underline{\textbf{type}}$: $find\text{-}eq\text{-}rule : RULE \to SSET \times RULE$
```

Annotations to *find-eq-rule*:

.1 − .2 Search for *equal* in the *body*, the *head* is unchanged.

End of annotations

```
3.      $find\text{-}eq\text{-}body(body_1) \triangleq$
 .1        $body_1 = \{\ \} \to (\{\ \},\{\ \}),$
 .2        $\mathsf{T} \to$
 .3           $\underline{\textbf{let}}$ $l \in body_1$ $\underline{\textbf{in}}$
 .4              $\underline{\textbf{cases}}$ $l$ :
 .5                 $\underline{\textbf{mk-}}POS(atom) \to$
 .6                    $\underline{\textbf{let}}$ $(set,body_2) = find\text{-}eq\text{-}atom(atom)$ $\underline{\textbf{in}}$
 .7                       $(set,body_2 \cup (body_1 \setminus \{l\}))$
 .8                 $\mathsf{T} \to$
 .9                    $\underline{\textbf{let}}$ $(set,body_2) = find\text{-}eq\text{-}body(body_1 \setminus \{l\})$ $\underline{\textbf{in}}$
 .10                      $(set,\{l\} \cup \{body_2\})$
3.11    $\underline{\textbf{type}}$: $find\text{-}eq\text{-}body : BODY \to SSET \times BODY$
```

Annotations to *find-eq-body*:

.1 − .10 We search for positive equal atoms.

End of annotations

```
4.    find-eq-atom(atom) ≜
 .1      cases atom :
 .2         mk-EQUAL(e,t) →
 .3            cases e :
 .4               mk-VAR()   →({ mk-BINDING(e,t) },{ }),
 .5               mk-CON(x) →
 .6                  cases t :
 .7                     mk-VAR()   → ({ mk-BINDING(t,e) },{ }),
 .8                     mk-CON(y) →
 .9                        x=y → ({ },{ }),
 .10                       T    → ({ },{ mk-POS(mk-HEAD(fail,{ })) }) ,
 .11                    T             → ({ },{ }) ,
 .12              T             → ({ },{ }) ,
 .13        T                     → ({ },{ mk-POS(atom) })
4.14  type: find-eq-atom : ATOM → SSET × BODY
```

Annotations to *find-eq-atom*:

.1 − .15 If we find an *equal* atom with at least one variable then we return a substitution set with the pair of terms to be substituted and an empty body, which means that the equal atom is removed from the rule body. Otherwise the body is unchanged.

End of annotations

C.3 Introduce anonymous variables

Auxiliary Domains

1. $OMAP \; = \; VAR \underset{m}{\rightarrow} \mathsf{N}_1$

```
2.    intro-av-db(db) ≜
 .1      db = { } → { },
 .2      T         →
 .3         let r₁ ∈ db in
 .4            let r₂ = intro-av-rule(r₁) in
 .5               { r₂ } ∪ intro-av-db(db \ { r₁ })
2.6   type: intro-av-db : DATABASE → DATABASE
```

Annotations to *intro-av-db*:

.1 − .4 The transformation is local to the *rules*, so they are treated one by one.

End of annotations

```
3.    intro-av-rule(rule) $\triangleq$
 .1     let map = count-rule(rule) in
 .2       let set = <mk-BINDING(v,ANONYMOUS VARIABLE)
 .3            | map(v) = 1>in
 .4           $\mathcal{S}$-set(set,rule)
3.5   type: intro-av-rule : RULE $\rightarrow$ RULE
```

Annotations to *intro-av-rule*:

.1 Count the occurrences of the variables in the rule

.2 Select all those that only occur once, and pair them with an anonymous variable

.3 Substitute all single variables with an anonymous variable

End of annotations

C.4 Auxiliary Functions

C.4.1 Substitute Terms

```
1.    $\mathcal{S}$-set(set,$rule_1$) $\triangleq$
 .1     set = { } $\rightarrow$ $rule_1$,
 .2     T         $\rightarrow$
 .3       let b $\in$ set in
 .4         let $rule_2$ = $\mathcal{S}$-RULE(b,$rule_1$) in
 .5           $\mathcal{S}$-set(set \ { b },$rule_2$)
1.6   type: $\mathcal{S}$-set : SSET $\times$ RULE $\rightarrow$ RULE
```

Annotations to $\mathcal{S}$-*set*:

.1 – .5 Substitute one binding ad a time.

End of annotations

C.4.2 Count Occurrences of Variables

```
1.    count-rule(mk-RULE(head,body)) $\triangleq$
 .1     let map = count-atom(head,[]) in
 .2         count-body(body,map)
1.3   type: count-rule : RULE $\rightarrow$ OMAP
```

Annotations to *count-rule*:

.1 – .2 Count variable occurrences in both the head and the body

End of annotations

```
2.      $count\text{-}body(body,map) \triangleq$
 .1       $body = \{\ \} \rightarrow map,$
 .2       $\top \qquad\qquad \rightarrow$
 .3         let $l \in body$ in
 .4           let $map_1 = count\text{-}literal(l,map)$ in
 .5             $count\text{-}body(body \setminus \{l\},\ map_1)$
2.6     type: $count\text{-}body : BODY \times OMAP \rightarrow OMAP$
```

Annotations to *count-body*:

.1 – .5 Count variable occurrences in one literal at a time

End of annotations

```
3.      $count\text{-}literal(l,map) \triangleq$
 .1       cases $l$ :
 .2         mk-$POS(atom) \rightarrow count\text{-}atom(atom,map),$
 .3         mk-$NEG(atom) \rightarrow count\text{-}atom(atom,map)$
3.4     type: $count\text{-}literal : LITERAL \times OMAP \rightarrow OMAP$
```

Annotations to *count-literal*:

.1 – .3 Number of occurrences are independent of atom sign

End of annotations

```
4.      $count\text{-}atom(atom,map) \triangleq$
 .1       cases $atom$ :
 .2         mk-$HEAD(,ts) \qquad\qquad \rightarrow$
 .3           $count\text{-}termlist(ts,map),$
 .4         mk-$LESS\text{-}THAN(e_1,e_2) \quad \rightarrow$
 .5           let $map_1 = count\text{-}expr(e_1,map)$ in
 .6             $count\text{-}expr(e_2,map_1),$
 .7         mk-$LESS\text{-}EQUAL(e_1,e_2) \rightarrow$
 .8           let $map_1 = count\text{-}expr(e_1,map)$ in
 .9             $count\text{-}expr(e_2,map_1),$
 .10        mk-$NOT\text{-}EQUAL(e_1,e_2) \rightarrow$
 .11          let $map_1 = count\text{-}expr(e_1,map)$ in
 .12            $count\text{-}expr(e_2,map_1),$
 .13        mk-$EQUAL(e_1,e_2) \qquad \rightarrow$
 .14          let $map_1 = count\text{-}expr(e_1,map)$ in
 .15            $count\text{-}expr(e_2,map_1)$
4.16    type: $count\text{-}atom : ATOM \times OMAP \rightarrow OMAP$
```

Annotations to *count-atom*:

.1 – .15 Count variable occurrences in atoms

End of annotations

```
5.    count-termlist(ts,map) ≜
 .1      ts = <> → map,
 .2      T        →
 .3         let map₁ = count-expr(hd ts,map) in
 .4            count-termlist(tl ts,map₁)
5.5   type: count-termlist : TERMLIST × OMAP → OMAP
```

Annotations to *count-termlist*:

.1 − .4 Count each term in the list

End of annotations

```
6.    count-expr(e,map) ≜
 .1      cases e :
 .2         mk-PLUS(e₁,e₂) →
 .3            let map₁ = count-expr(e₁,map) in
 .4               count-expr(e₂,map₁),
 .5         mk-MULT(e₁,e₂) →
 .6            let map₁ = count-expr(e₁,map) in
 .7               count-expr(e₂,map₁),
 .8         mk-SIGN(e₁)       → count-expr(e₁,map),
 .9         mk-VAR()          → count-var(e,map),
 .10        T                 → map
6.11  type: count-expr : TERM × OMAP → OMAP
```

Annotations to *count-expr*:

.1 − .10 During the optimization we do not distinguish between *terms* and expressions.

End of annotations

```
7.    count-var(v,map) ≜
 .1      v ∈ dom map → map + [ v ↦ map(v) + 1],
 .2      T           → map ∪ [ v ↦ 1]
7.3   type: count-var : TERM × OMAP → OMAP
```

Annotations to *count-var*:

.1 If the variable has been counted before then increase the number of occurrences.

.2 Otherwise introduce the variable to the map.

End of annotations

Appendix D

Recursive Query Subquery

This specification of QSQR is more detailed than the one presented in chapter 7.4 and we have implemented our interpreter from this specification. Not all auxiliary functions have been specified this is also the case for the important functions *join* and *difference*. However, the implementation can be studied in section G.7.1 and the theori can be studied in [Ullman 89a].

D.1 Domains and Conditions

The abstract domain used in the interpreter to exchange information and to carry the answers is a *relation* (R) and we need to check whether a query is new or have been asked before (*NEW-QUERY*).

Abstract Domains

1. *NEW-QUERY* $=$ *(ID* $\times$ *TSS)***-set**

Since we intend to use the specification as work schedule, some of the information is stored in an external database. This is the case for the extensional database (*EDB*), the rulegraph (*RULEGRAPH*), the range restriction information (*RANGE*), the adornment information (*ADORN*) and information on what queries that have been asked and may not be asked again (*NEW-QUERY*).

When we consult or change the external database during the following specification we will use the names given above extended with a *-db* e.g. *edb-db*. The result of this is that the five variables in this external database are not "carried around" as parameters in the specification and thereby give a more easy-to-overview specification.

D.2 Interpreter

Transform the query to a *generalized query* [Ceri 90]. The generalized query can contain more than one constant value for each argument which is necessary when one subquery passes information on to the next subquery. The DATALOG database is contained in the external database variables *EDB* and *RULEGRAPH*.

```
2.     interpreter(query) ≜
 .1       let mk-QUERY(mk-POS(mk-HEAD(qid,qts))) = query in
 .2       let rts = convert-qtermlist(qts) in
 .3       let ts = pre-qsqr(rts) in
 .4          qsqr(mk-R(qid,rts,{ ts }))
2.5    type: interpreter : QUERY → R
```

Annotations to *interpreter*:

.2 Translate the QTERMLIST to a TERMLIST

.3 Information on constants and anonymous variables is extracted.

.4 Call RECURSIVE QUERY SUBQUERY with this relation.

End of annotations

D.3 Recursive Query Subquery

Our algorithm is based on the *recursive query subquery* algorithm presented in [Ceri 90, pp. 155-160]. The algorithm is a realization of the top-down depth-first resolution strategy *SLD-AL resolution* presented by [Vieille 89].

We have extended the algorithm to handle eventual manually specified range restrictions *within-range* and AL technique, i.e. *Admissibility test* and *Lemma resolution.*

```
3.    qsqr(rel_1) ≜
 .1     let mk-R(id,termlist) = rel_1 in
 .2       mk-INIT(id) ∈ rulegraph-db → mk-R(id,termlist,{ })
 .3       T                          →
 .4         let rel_2 = within-range(rel_1) in
 .5         let rel_3 = consult-edb(rel_2) in
 .6         let rel_4 = within-range(rel_3) in
 .7         let group_1 = non-recursive-group(id) in
 .8         let rel_5 = consult-idb(rel_2,group_1) in
 .9         let rel_6 = within-range(rel_5) in
 .10        let edb-db = store-lemma(rel_6,edb-db) in
 .11        let group_2 = recursive-group(id) in
 .12        let rel_7 = consult-recursive(rel_2,group_2) in
 .13        let rel_8 = within-range(rel_7) in
 .14        let rel_9 = relation-union(rel_4,rel_6) in
 .15          relation-union(rel_8,rel_9)
3.16  type: qsqr : R → R
```

Annotations to *qsqr*:

.2 If *INIT(id)* is a member of the *rulegraph-db* then there exist no rules of that name and the empty set is returned.

.4,6,9,13 *within-range* takes out those tuples that are not within a specified range.

.5 The extensional database *edb-db* is consulted.

.7 *group*$_1$ consists of all non-recursive rules of name *id*.

.8 Consult these rules.

.10 The answers are added to the already known facts in *edb-db*.

.11−.13 Consult the recursive rules of name *id*.

.14−.15 The answers from the *facts* and the *rules* are added together with a union.

End of annotations

D.4 Consult the Extensional Database

The extensional database *EDB* consists of facts, i.e. rules without body and only constants as arguments in the header.

```
4.    consult-edb(**mk-**R(id,ts,tss_1) ≜
 .1     let tss_2 = if id ∈ dom edb-db
 .2           then edb-db(id)
 .3           else { } in
 .4         let tss_3 = match(tss_1,tss_2) in
 .5           let (tss_4,) = filter(ts,tss_3,ts) in
 .6             mk-R(id,ts,tss_4)
4.7   type: consult-edb : R → R
```

Annotations to *consult-edb*:

.1 − .6 If there are any facts then match them with the generalized query and filter those facts out that do not unify with the termlist *ts*.

End of annotations

D.5 Consult the Intensional Database

The intensional database, at this moment represented by the rulegraph, consists of both recursive and non-recursive rules.

D.5.1 Consult Non-recursive Rules

```
5.    consult-idb(rel_1,group) ≜
 .1     let mk-R(id,ts,) = rel_1 in
 .2         group = { } → mk-R(id,ts,{ })
 .3         T           →
 .4           let rule ∈ group in
 .5             let rel_2 = consult-rule(rel_1,rule) in
 .6               let rel_3 = consult-idb(rel_1,group \ { rule }) in
 .7                 relation-union(rel_2,rel_3)
5.8   type: consult-idb : R × IDB → R
```

Annotations to *consult-idb*:

.2 If the group is empty the return a fail.

.3 − .7 Else query the rules one by one and union the answers. Note that the fail relation is an identity for the *relation-union* function.

End of annotations

D.5.2 Consult Recursive Rules

```
6.    consult-recursive(rel_1,group) ≜
 .1     let mk-R(id,ts,) = rel_1 in
 .2       let dummy = mk-R(di,< >,{ }) be s.t. id ≠ di in
 .3         group = { }             → mk-R(id,ts,{ })
 .4         (id,tss) ∈ new-query-db → mk-R(id,ts,{ })
 .5         T                       →
 .6           let new-query-db = new-query-db ∪ (id,tss) in
 .7           let rel_2 = consult-idb(rel_1,group) in
 .8           let edb-db = store-lemma(rel_2,edb-db) in
 .9             consult-recursive2(dummy,rel_2,group)
6.10  type: consult-recursive : R × IDB → R
```

Annotations to *consult-recursive*:

.2 If the group is empty the return a fail.

.3 If the query is not new then stop the recursion.

.4 − .8 Otherwise register the query as new and consult the recursive rules.

End of annotations

```
7.    consult-recursive2(rel_1,rel_2,group) ≜
 .1     rel_1 = rel_2 → rel_2,
 .2     T             → consult-recursive3(rel_2,group)
7.3   type: consult-recursive2 : R × R × IDB → R
```

Annotations to *consult-recursive2*:

.1 The dynamic halting condition. If we have reached a fixed point, then stop the recursion.

.2 Else continue.

End of annotations

```
8.    consult-recursive3(rel_1,group) ≜
 .1     let rel_2 = consult-idb(rel_1,group) in
 .2     let edb-db = store-lemma(rel_2,edb-db) in
 .3       consult-recursive2(dummy,rel_2,group)
8.4   type: consult-recursive : R × IDB → R
```

Annotations to *consult-recursive*:

.1 − .3 Consult the recursive rules until the fixed point is reached.

End of annotations

D.6 Consult a Rule

D.6.1 consult-rule

```
9.    $consult\text{-}rule(rel_1,rule) \triangleq$
 .1      **let** **mk-**$R(id,qts_1,qtss_1) = rel_1$,
 .2          **mk-**$RULE($**mk-**$HEAD(id,rts_1),body_1)$ **in**
 .3        **let** $varset = all\text{-}var\text{-}termlist(rts_1)$ **in**
 .4        **let** $qts_2 = rename(varset,qts_1)$ **in**
 .5        **let** $rtss_1 = filter(qts_2,qtss_1,rts_1)$ **in**
 .6        **let** $\theta = mgu(\{\ \},rts_1,qts_2)$ **in**
 .7        **if** $\theta \neq \triangledown$
 .8        **then** **let** $rts_2 = \mathcal{S}\text{-}TERMLIST(rts_1,\theta)$ **in**
 .9             **let** $body_2 = \mathcal{S}\text{-}BODY(body_1,\theta)$ **in**
 .10            **let** $(rts_3,rtss_3) = pre\text{-}loop(rts_2,rtss_1,body_2)$ **in**
 .11            **let** **mk-**$R(id,ts,tss) = loop($**mk-**$R(id,rts_3,rtss_3),body_2)$ **in**
 .12            **let** $tss_1 = project\text{-}tuples(ts,tss,rts_1)$ **in**
 .13              **mk-**$R(id,qts_1,tss_2)$
 .14       **else** **mk-**$R(id,qts_1,\{\ \})$
9.15  **type**: $consult\text{-}rule : R \times RULE \to R$
```

Annotations to *consult-rule*:

.3 − .4 Before unifying the query with the rule we must ensure that the query and the rule header do not have variables in common. This is done by renaming variables in the query that also occur in the rule header.

.5 Then filter out those tuples in the generalized query that do not unify with the rule.

.6 Try to compute the *most general unifier* (**mgu**) for query and the rule.

.7 − .9 If the **mgu** θ exists ($\triangledown$ means that the query and the rule header can not be unified) then substitute the rule header rts_1 and the rule body $body_1$ with θ to correct for local bindings in the query, i.e. queries like ?− p(Z,Z). to rules like p(X,Y) :− r(X,Y)..

.10 *pre-loop* forms a new rule-relation, where all variables in the rule are in rts_3. It is important that all variables are present in the generalized query, for the arithmetical operations to give uninstantiated variables a value.

.11 The *loop* function will return the rule-relation instantiated with the values from the body.

.12 The answer tuples are projected on to the rule head rts_1.

.13 And finally the answer is returned.

.14 If the query and the rule do not unify, i.e. a **mgu** does not exist, the return the fail relation.

End of annotations

D.6.2 loop

```
10.   $loop(rel_1,body_1) \triangleq$
 .1      $body = \underline{\textbf{-set}}\,\} \rightarrow rel_1,$
 .2      T                $\rightarrow$
 .3         $\underline{\textbf{let}}$ $\underline{\textbf{mk-}}R(,rts,rtss) = rel_1$ $\underline{\textbf{in}}$
 .4         $\underline{\textbf{let}}$ $bindings = binding\text{-}info(rtss)$ $\underline{\textbf{in}}$
 .5         $\underline{\textbf{let}}$ $(literal,body) = selection\text{-}function(rts,bindings,body_1)$$\underline{\textbf{in}}$
 .6         $\underline{\textbf{let}}$ $rel_2 = cases\text{-}literal(literal,rel_1)$ $\underline{\textbf{in}}$
 .7            $loop(rel_2,body)$
10.8  $\underline{\textbf{type}}$: $loop : R \times BODY \rightarrow R$
```

Annotations to *loop*:

.1 When there are no more literals left stop the *loop* and return the answer relation.

.2 Otherwise

.3−.4 The information on what variables that are instantiated can be determined in two ways: one way is by looking at the adornment for each literal that have been selected and from that determine what variables that can be *guaranteed* bound and what variables that can not be *guaranteed* bound, another way is to actually check all variables in the relation to see whether they are bound or free. We have done it the later way, because variables that can not be *guaranteed* bound may actually be bound any way and thereby enable a larger set of literals to be ready for selection!

.5 Then select a literal/subgoal

.6 for execution

.7 and continue the *loop*.

End of annotations

D.6.3 cases-literal

```
11.     $cases\text{-}literal(literal, rel_1) \triangleq$
  .1      **cases** $literal$ :
  .2        **mk-**$POS(atom) \rightarrow$
  .3          **cases** $atom$ :
  .4            **mk-**$HEAD() \rightarrow$
  .5              **let** $rel_2 = pre\text{-}call(rel_1, atom)$ **in**
  .6                **let** $rel_3 = qsqr(rel_2)$ **in**
  .7                  $join(rel_1, rel_3)$
  .8            T              $\rightarrow$
  .9              **let** **mk-**$R(id, ts, tss_1) = rel_1$ **in**
  .10               **let** $tss_2 = select\text{-}tuples(ts, tss_1, atom)$ **in**
  .11                 **mk-**$R(id, ts, tss_2)$
  .12       **mk-**$NEG(atom) \rightarrow$
  .13         **let** $rel_2 = pre\text{-}call(rel_1, atom)$ **in**
  .14           **let** $rel_3 = qsqr(rel_2)$ **in**
  .15             $diff(rel_1, rel_3)$
11.16   **type**: $cases\text{-}literal : LITERAL \times R \rightarrow R$
```

Annotations to *cases-literal*:

.1 The literal is either a positive or a negative atom.

.2 If it is positive it is either a user-predicate or a standard-predicate.

.3 − .7 If it is a user predicate the call the *recursive query subquery* algorithm recursively.

.8 − .11 Otherwise let the internal standard predicates perform a selection on the relation.

.12 − .15 In the negative call we say that all tuples that can be proved are subtracted from the relation by the difference operation. Contrary to the positive call where all proved tuples were added by the join operation.

End of annotations

D.6.4 select-tuples

```
12.     $select\text{-}tuples(ts, tss, atom) \triangleq$
  .1      $tss = \{\ \} \rightarrow \{\ \},$
  .2      T           $\rightarrow$
  .3        **let** $tp \in tss$ **in**
  .4        **let** $tss_1 = select\text{-}tuple(ts, tp, atom)$ **in**
  .5        **let** $tss_2 = select\text{-}tuples(ts, tss \setminus \{ tp \}, atom)$ **in**
  .6          $tss_1 \cup tss_2$
12.7    **type**: $select\text{-}tuples : TERMLIST \times TSS \times ATOM \rightarrow TSS$
```

Annotations to *select-tuples*:

.1 − .6 Test the tuples one by one, those which survive the test are returned.

End of annotations

D.6.5 select-tuple

```
13.     $select\text{-}tuple(ts,tp,atom) \triangleq$
.1        cases $atom$ :
.2          mk-$LESS\text{-}THAN(t_1,t_2) \rightarrow$
.3            let $v_1 = read\text{-}value(t_1,ts,tp),\ v_2 = read\text{-}value(t_2,ts,tp)$ in
.4              if $less\text{-}than(v_1,v_2)$ then $\{tp\}$ else $\{\ \}$
.5          mk-$LESS\text{-}EQUAL(t_1,t_2) \rightarrow$
.6            let $v_1 = read\text{-}value(t_1,ts,tp),\ v_2 = read\text{-}value(t_2,ts,tp)$ in
.7              if $less\text{-}equal(v_1,v_2)$ then $\{tp\}$ else $\{\ \}$
.8          mk-$NOT\text{-}EQUAL(t_1,t_2) \rightarrow$
.9            let $v_1 = read\text{-}value(t_1,ts,tp),\ v_2 = read\text{-}value(t_2,ts,tp)$ in
.10             if $not\text{-}equal(v_1,v_2)$ then $\{tp\}$ else $\{\ \}$
.11         mk-$EQUAL(expr,t_3) \rightarrow$
.12           let $v_3 = read\text{-}value(t_3,ts,tp)$ in
.13             cases $expr$ :
.14               mk-$PLUS(t_1,t_2) \rightarrow$
.15                 let $v_1 = read\text{-}value(t_1,ts,tp),\ v_2 = read\text{-}value(t_2,ts,tp)$ in
.16                   $v_1 =$ mk-$CON() \wedge v_2 =$ mk-$CON() \rightarrow$
.17                     let $v = plus(v_1,v_2)$ in
.18                       if $match(v,v_3)$ then $\{write(v_3,t_3,ts,tp)\}$ else $\{\ \}$
.19                   $v_1 =$ mk-$CON() \wedge v_3 =$ mk-$CON() \rightarrow$
.20                     let $v = minus(v_3,v_1)$ in
.21                       if $match(v,v_2)$ then $\{write(v_2,t_2,ts,tp)\}$ else $\{\ \}$
.22                   $v_2 =$ mk-$CON() \wedge v_3 =$ mk-$CON() \rightarrow$
.23                     let $v = minus(v_3,v_2)$ in
.24                       if $match(v,v_1)$ then $\{write(v_1,t_1,ts,tp)\}$ else $\{\ \}$
.25               mk-$MULT(t_1,t_2) \rightarrow$
.26                 let $v_1 = read\text{-}value(t_1,ts,tp),\ v_2 = read\text{-}value(t_2,ts,tp)$ in
.27                   $v_1 =$ mk-$CON() \wedge v_2 =$ mk-$CON() \rightarrow$
.28                     let $v = times(v_1,v_2)$ in
.29                       if $match(v,v_3)$ then $\{write(v_3,t_3,ts,tp)\}$ else $\{\ \}$
.30                   $v_1 =$ mk-$CON() \wedge v_3 =$ mk-$CON() \rightarrow$
.31                     let $v = divide(v_3,v_1)$ in
.32                       if $match(v,v_2)$ then $\{write(v_2,t_2,ts,tp)\}$ else $\{\ \}$
.33                   $v_2 =$ mk-$CON() \wedge v_3 =$ mk-$CON() \rightarrow$
.34                     let $v = divide(v_3,v_2)$ in
.35                       if $match(v,v_1)$ then $\{write(v_1,t_1,ts,tp)\}$ else $\{\ \}$
.36               mk-$SIGN(t_1) \rightarrow$
.37                 let $v_1 = read\text{-}value(t_1,ts,tp)$ in
.38                   $v_1 =$ mk-$CON() \rightarrow$ let $v = minus(zero,v_1)$ in
.39                     if $match(v,v_1)$ then $\{write(v_1,t_1,ts,tp)\}$ else $\{\ \}$
.40                   $v_3 =$ mk-$CON() \rightarrow$ let $v = minus(zero,v_3)$ in
.41                     if $match(v,v_3)$ then $\{write(v_3,t_3,ts,tp)\}$ else $\{\ \}$
13.42   type: $select\text{-}tuple : TERMLIST \times TERMLIST \times ATOM \rightarrow TSS$
```

Annotations to *select-tuple*:

.0 *select-tuple* contains all standard predicates.

.1 – .41 If the tuple *tp* matches the constraint *atom*, then it is returned. Otherwise not. If the *atom* is an arithmetical operation, then the anonymous variable in the tuple is replaced with the computed value.

End of annotations

D.7 Within-range

```
14.    $within\text{-}range(\underline{\mathbf{mk\text{-}}}R(id,termlist,tss_1)) \triangleq$
  .1     $id \notin \underline{\mathbf{dom}} range\text{-}db \rightarrow \underline{\mathbf{mk\text{-}}}R(id,termlist,tss_1),$
  .2     $\mathsf{T} \qquad\qquad\qquad \rightarrow$
  .3       $\underline{\mathbf{let}}\ tss_2 = \{\ ts \mid ts \in tss_1 \wedge$
  .4           $within\text{-}range\text{-}tuple(1,id,ts)\ \}\ \underline{\mathbf{in}}$
  .5         $\underline{\mathbf{mk\text{-}}}R(id,termlist,tss_2)$
14.6   $\underline{\mathbf{type}}: within\text{-}range : R \rightarrow R$
```

Annotations to *within-range*:

.1 If there is no range restrictions specified for this predicate then all tuples are accepted.

.2 – .5 Else test each tuple for being within the domain.

End of annotations

```
15.    $within\text{-}range\text{-}tuple(pos,id,ts) \triangleq$
  .1     $ts = <> \qquad\qquad\qquad \rightarrow \underline{\mathbf{true}},$
  .2     $pos \notin \underline{\mathbf{dom}} range\text{-}db(id) \rightarrow$
  .3       $within\text{-}range\text{-}tuple(pos{+}1,id,\underline{\mathbf{tl}}ts),$
  .4     $\mathsf{T} \qquad\qquad\qquad\qquad \rightarrow$
  .5       $\underline{\mathbf{let}}\ (low,hi) = range\text{-}db(id)(pos)\ \underline{\mathbf{in}}$
  .6         $\underline{\mathbf{cases}}\ \underline{\mathbf{hd}} ts :$
  .7           $\underline{\mathbf{mk\text{-}}}CON() \rightarrow$
  .8             $less\text{-}equal(low,\underline{\mathbf{hd}}ts) \wedge$
  .9             $less\text{-}equal(\underline{\mathbf{hd}}ts,hi) \wedge$
  .10            $within\text{-}range\text{-}tuple(pos{+}1,id,\underline{\mathbf{tl}}ts),$
  .11          $\mathsf{T} \qquad \rightarrow$
  .12            $within\text{-}range\text{-}tuple(pos{+}1,id,\underline{\mathbf{tl}}ts)$
15.13  $\underline{\mathbf{type}}: within\text{-}range\text{-}tuple : \mathsf{N}_1 \times ID \times TERMLIST \rightarrow \mathsf{BOOL}$
```

Annotations to *within-range-tuple*:

.1 The recursion ends with the empty tuple

.2 – .3 If there is no range restrictions specified for this argument then check the next argument.

.4−.12 Else check that the front argument is within the specified range.

End of annotations

D.8 Store Lemma

```
16.     store-lemma(mk-R(id,,tss),edb) ≜
  .1      id ∈ dom edb →
  .2        edb + [ id ↦ edb(id) ∪ tss ],
  .3      T              →
  .4        edb ∪ [ id ↦ tss ]
16.5    type: store-lemma : R × EDB → EDB
```

Annotations to *store-lemma*:

.0 The new facts are called *lemmas* because they have been proved by the interpreter.

.1−.4 Add the facts from *tss* to the extensional database.

End of annotations

D.9 Non-recursive group

```
17.     non-recursive-group(id) ≜
  .1      { rule |
  .2      let rule = mk-RULE(mk-HEAD(id,),) in
  .3        rule ∈ group ∧ mk-STEP(group) ∈ rulegraph-db}
17.4    type: non-recursive-group : ID → IDB
```

Annotations to *non-recursive-group*:

.1−.3 Select all non-recursive rules with the name *id* from the *RULE-GRAPH.*

End of annotations

D.10 Recursive group

```
18.     recursive-group(id) ≜
  .1      { rule |
  .2      let rule = mk-RULE(mk-HEAD(id,),) in
  .3        rule ∈ group ∧ mk-REPEAT(group) ∈ rulegraph-db}
18.4    type: recursive-group : ID → IDB
```

Annotations to *recursive-group*:

.1 − .3 Select all recursive rules with the name *id* from the *RULEGRAPH*.

End of annotations

D.11 Selection Function

D.11.1 Auxiliary Domains

```
19.    $BL = BF^*$
20.    $BF = \text{FREE} \mid \text{BOUND}$
```

D.11.2 Selection Function

```
21.    $selection\text{-}function(ts,binding,body,\theta) \triangleq$
  .1     $\underline{\textbf{let}}\ body_1 = \{\, l \mid l \in body \wedge ready(l,ts,binding,adorn\text{-}db) \,\}\ \underline{\textbf{in}}$
  .2     $\underline{\textbf{let}}\ \beta\text{-}map = [\, l \mapsto beta(l,edb\text{-}db,ts,binding) \mid l \in body_1 \,]\ \underline{\textbf{in}}$
  .3     $\underline{\textbf{let}}\ max\text{-}\beta = max\{\, \underline{\textbf{rng}}\beta\text{-}map \,\}\ \underline{\textbf{in}}$
  .4     $\underline{\textbf{let}}\ body_2 = \{\, l \mid l \in body_1 \wedge \beta\text{-}map(l) = max\text{-}\beta \,\}\ \underline{\textbf{in}}$
  .5     $\underline{\textbf{let}}\ selected\text{-}literal \in body_2\ \underline{\textbf{in}}$
  .6        $(selected\text{-}literal, body_2 \setminus \{\, selected\text{-}literal \,\})$
21.7   $\underline{\textbf{type}}$: $selection\text{-}function : TERMLIST \times BL \times EDB$
  .8           $\times\ ADORNMENT \times BODY \times SUBSTITUTION$
  .9           $\to LITERAL \times BODY$
```

Annotations to *selection-function*:

.1 Only consider those literals that are ready to be selected.

.2 Calculate β for those that are left.

.3 − .6 Select the literal between those literals with maximum β.

End of annotations

D.11.3 Rename

```
22.    $rename(varset,ts) \triangleq$
  .1     let $t \in$ TOKEN be s.t. $t \notin varset$ in
  .2     let $v = $ mk-$VAR(t)$ in
  .3     let $varset_1 = varset \cup \{v\}$ in
  .4       $ts = <> \rightarrow <>$,
  .5       T          $\rightarrow$
  .6         hd$ts \in varset \rightarrow$
  .7           let $ts_1$ $\mathcal{S}$-$TERMLIST($tl$ts,\{$ mk-$BINDING($hd$ts,v)\})$ in
  .8             $v$ ^ $rename(varset_1,ts_1)$
22.9   type: $VARSET \times TERMLIST \rightarrow TERMLIST$
```

Annotations to *rename*:

.0 The termlist *ts* must not have variables in common with the *varset*.

End of annotations

D.11.4 Filter

```
23.    $filter(qts,qtss,rts) \triangleq$
  .1     $qtss = \{\ \} \rightarrow \{\ \}$,
  .2     T          $\rightarrow$
  .3       let $ts \in qtss$ in
  .4         let $gts = form\text{-}query(qts,ts)$ in
  .5         let $\theta = mgu(\{\ \},rts,gts)$ in
  .6           $\theta = \bigtriangledown \rightarrow filter(qtss \setminus \{qts\},rts)$,
  .7           T       $\rightarrow$
  .8             let $gts_1 = \mathcal{S}$-$TERMLIST(gts,\theta)$ in
  .9               $gts_1 \cup filter(qtss \setminus \{qts\},rts)$
23.10  type: $filter : TSS \times TERMLIST \rightarrow TSS$
```

Annotations to *filter*:

.0 Only termlists from the generalized query that unifies with rule termlist are returned.

End of annotations

D.11.5 Ready

```
24.    ready(l,ts,bind,adorn) ≜
  .1     cases l :
  .2       mk-NEG(atom) → ready-neg(atom,ts,bind),
  .3       mk-POS(atom) → ready-pos(atom,ts,bind,adorn)
24.4   type: ready : LITERAL × TERMLIST × BL
  .5               × ADORNMENT → BOOL
```

Annotations to *ready*:

.0 Determine whether the literal l is ready to be evaluated or not.

End of annotations

```
25.    ready-neg(mk-HEAD(,ts),termlist,binding) ≜
  .1     arg-bound(ts,termlist,binding) = lents
25.2   type: ready-neg : ATOM × TERMLIST × BL → BOOL
```

Annotations to *ready-neg*:

.1 All arguments must be BOUND for a negative literal to be ready.

End of annotations

```
26.    ready-pos(atom,ts,binding,adorn) ≜
  .1     cases atom :
  .2       mk-LESS-THAN($t_1$,$t_2$) → arg-bound(<$t_1$,$t_2$>,ts,binding) = 2,
  .3       mk-LESS-EQUAL($t_1$,$t_2$)→ arg-bound(<$t_1$,$t_2$>,ts,binding) = 2,
  .4       mk-NOT-EQUAL($t_1$,$t_2$)→ arg-bound(<$t_1$,$t_2$>,ts,binding) = 2,
  .5       mk-EQUAL(e,$t_3$)        →
  .6         cases e :
  .7           mk-PLUS($t_1$,$t_2$)  → arg-bound(<$t_1$,$t_2$,$t_3$>,ts,binding) ≥ 2,
  .8           mk-MULT($t_1$,$t_2$) → arg-bound(<$t_1$,$t_2$,$t_3$>,ts,binding) ≥ 2,
  .9           mk-SIGN($t_1$       → arg-bound(<$t_1$,$t_3$>,ts,binding) ≥ 1 ,
  .10      mk-HEAD(id,$ts_1$)       →
  .11        let bfs = adorn(id) in
  .12          (∃ bf ∈ bfs)(ready-termlist($ts_1$,ts,binding,bf))
26.13  type: ready-pos :  ATOM × TERMLIST × BL
  .14                    × ADORNMENT → BOOL
```

Annotations to *ready-pos*:

.2 – .4 All arguments must be BOUND for a comparative literal to be ready.

.5 – .9 Maximum one free argument for an arithmetic literal to be ready.

.10 – .12 The adornment of the predicate determines whether the predicate is ready or not.

End of annotations

```
27.   ready-termlist(ts_1,ts_2,binding,bf) ≜
  .1     ts_1 = <> → true,
  .2     T         →
  .3        ready-term(hd ts_1,ts_2,binding,hd bf) ∧
  .4        ready-termlist(tl ts_1,ts_2,binding,tl bf)
27.5  type: ready-termlist : TERMLIST × TERMLIST × BL
  .6                         × PATTERN → BOOL
```

Annotations to *ready-termlist*:

.1−.4 Test one term at a time.

End of annotations

```
28.   ready-term(t,ts,binding,bf) ≜
  .1     cases t :
  .2        ANONYMOUS VARIABLE → true,
  .3        mk-CON()           → true,
  .4        mk-VAR()           →
  .5           cases var-bound(t,ts,binding) :
  .6              0 → bf ≠ BB,
  .7              1 → true
28.8  type: ready-term : TERM × TERMLIST × BL × PATTERN → BOOL
```

Annotations to *ready-term*:

.2 It is assumed that the adornment *bf* never can be demanded PRE-BOUND (BB) when an anonymous variable occurs here, be cause of the range restriction demand.

.3 Constants matches any demand.

.6 If a variable is FREE, but demanded BOUND then it is not ready,

.7 otherwise it ready.

End of annotations

D.11.6 The Degree of Binding Function BETA

```
29.     beta(literal,edb,termlist,binding) ≜
  .1      cases literal :
  .2        mk-NEG()      → 1,
  .3        mk-POS(atom) →
  .4          cases atom :
  .5            mk-HEAD(id,ts) →
  .6              let a = len ts,
  .7                  b = arg-bound(ts,termlist,binding),
  .8                  c = card(edb(id)) in
  .9                if a=0 ∨ c=0
  .10               then 1
  .11               else  b / (a ∗ c),
  .12           T                  → 1
29.13   type: beta : LITERAL × EDB × TERMLIST × BL → REAL
```

Annotations to *beta*:

.1 Beta is 1 for negative literals.

.9 Beta is 1 for comparative literals.

.2−.8 Otherwise compute the degree of binding.

End of annotations

D.11.7 Auxiliary Functions

Number of bound arguments

```
30.     arg-bound(ts1,ts2,binding) ≜
  .1      ts1 = < > → 0,
  .2      T         →
  .3        let rest = arg-bound(tl ts1,ts2,binding) in
  .4          cases hd ts1 :
  .5            mk-VAR(v) → var-bound(hd ts1,ts2,binding) + rest,
  .6            T         → rest
30.7    type: arg-bound : TERMLIST × TERMLIST × BL → N0
```

Annotations to *arg-bound*:

.0 Counts the number of bound variables in the termlist ts_1.

End of annotations

Check if a variable is bound or free

```
31.   $var\text{-}bound(v,ts,binding) \triangleq$
  .1    $ts = <> \rightarrow 0,$
  .2    $\top \qquad \rightarrow$
  .3      $v = \underline{\mathbf{hd}}ts \rightarrow$
  .4        $\underline{\mathbf{hd}}binding = \text{BOUND} \rightarrow 1,$
  .5        $\top \qquad\qquad \rightarrow 0,$
  .6      $\top \qquad \rightarrow var\text{-}bound(v,<ts,\underline{\mathbf{tl}}binding)$
31.7  $\underline{\mathbf{type}}: var\text{-}bound : VAR \times TERMLIST \times BL \rightarrow (\ 0 \mid 1)$
```

Annotations to *var-bound*:

.0 If the variable v occurs in *ts* and if it is BOUND then return 1 otherwise 0.

End of annotations

D.12 Substitution

```
32.   $\mathcal{S}\text{-}RULE(\underline{\mathbf{mk\text{-}}}RULE(head_1,body_1),\theta) \triangleq$
  .1    $\underline{\mathbf{let}}\ head_2 = \mathcal{S}\text{-}HEAD(head_1,\theta),$
  .2        $body_2 = \mathcal{S}\text{-}BODY(body_1,\theta)\ \underline{\mathbf{in}}$
  .3      $\underline{\mathbf{mk\text{-}}}RULE(head_2,body_2)$
32.4  $\underline{\mathbf{type}}: \mathcal{S}\text{-}RULE : RULE \times SUBSTITUTION \rightarrow RULE$
```

Annotations to $\mathcal{S}$-*RULE*:

.1 − .3 Substitute a rule by substituting the head and the body.

End of annotations

```
33.   $\mathcal{S}\text{-}BODY(body,\theta) \triangleq$
  .1    $body = \{\ \} \rightarrow \{\ \},$
  .2    $\top \qquad \rightarrow$
  .3      $\underline{\mathbf{let}}\ l \in body\ \underline{\mathbf{in}}$
  .4      $\underline{\mathbf{let}}\ l_1 = \mathcal{S}\text{-}LITERAL(l,\theta)\ \underline{\mathbf{in}}$
  .5      $\underline{\mathbf{let}}\ body_1 = \mathcal{S}\text{-}BODY(body \setminus \{\ l\ \},\theta)\ \underline{\mathbf{in}}$
  .6        $l_1 \cup body_1$
33.7  $\underline{\mathbf{type}}: \mathcal{S}\text{-}BODY : BODY \times SUBSTITUTION \rightarrow BODY$
```

Annotations to $\mathcal{S}$-*BODY*:

.1 − .6 Substitute the *literals* one by one.

End of annotations

```
34.    𝒮-LITERAL(l,θ) ≜
  .1     cases l :
  .2       mk-POS(atom) →
  .3         let atom₁ = 𝒮-ATOM(atom,θ) in
  .4           mk-POS(atom₁)
  .5       mk-NEG(atom) →
  .6         let atom₁ = 𝒮-ATOM(atom,θ) in
  .7           mk-NEG(atom₁)
34.8   type: 𝒮-LITERAL : LITERAL × SUBSTITUTION → LITERAL
```

Annotations to $\mathcal{S}$-*LITERAL*:

.1 − .7 There are *positive* and *negative* literals.

End of annotations

```
35.    𝒮-ATOM(atom,θ) ≜
  .1     cases atom :
  .2       mk-HEAD(id,ts)          →
  .3         let ts₁ = 𝒮-TERMLIST(ts,θ) in
  .4           mk-HEAD(id,ts₁)
  .5       mk-LESS-THAN(t₁,t₂)   →
  .6         let s₁ = 𝒮-TERM(t₁,θ),
  .7             s₂ = 𝒮-TERM(t₂,θ) in
  .8           mk-LESS-THAN(s₁,s₂),
  .9       mk-LESS-EQUAL(t₁,t₂) →
  .10        let s₁ = 𝒮-TERM(t₁,θ),
  .11            s₂ = 𝒮-TERM(t₂,θ) in
  .12          mk-LESS-EQUAL(s₁,s₂),
  .13      mk-NOT-EQUAL(t₁,t₂)  →
  .14        let s₁ = 𝒮-TERM(t₁,θ),
  .15            s₂ = 𝒮-TERM(t₂,θ) in
  .16          mk-NOT-EQUAL(s₁,s₂),
  .17      mk-EQUAL(e,t)          →
  .18        let s₁ = 𝒮-EXPR(e,θ),
  .19            s₂ = 𝒮-TERM(t,θ) in
  .20          mk-EQUAL(s₁,s₂)
35.21  type: 𝒮-ATOM : ATOM × SUBSTITUTION → ATOM
```

Annotations to $\mathcal{S}$-*ATOM*:

.1 − .8 Atoms are either *heads* or *comparative predicates.*

End of annotations

```
36.    $\mathcal{S}$-HEAD(mk-HEAD(p,ts),$\theta$) $\triangleq$
  .1      let $ts_1$ = $\mathcal{S}$-TERMLIST(ts,$\theta$) in
  .2        mk-HEAD(p,$ts_1$)
36.3   type: $\mathcal{S}$-HEAD : HEAD $\times$ SUBSTITUTION $\to$ HEAD
```

Annotations to $\mathcal{S}$-*HEAD*:

.1−.2 Substitute the *termlist.*

End of annotations

```
37.    $\mathcal{S}$-TERMLIST(ts,$\theta$) $\triangleq$
  .1      ts = < > $\to$ < >,
  .2      T        $\to$
  .3        let $ts_1$ = $\mathcal{S}$-TERMLIST(tl ts,$\theta$),
  .4            t = $\mathcal{S}$-TERM(hd ts,$\theta$) in t ^ $ts_1$
37.5   type: $\mathcal{S}$-TERMLIST :  TERMLIST $\times$ SUBSTITUTION $\to$
  .6                              TERMLIST
```

Annotations to $\mathcal{S}$-*TERMLIST*:

.1−.10 Substitute one *term* at a time.

End of annotations

```
38.    $\mathcal{S}$-EXPR(expr,$\theta$) $\triangleq$
  .1      cases expr :
  .2        mk-PLUS($t_1$,$t_2$) $\to$
  .3          let $s_1$ = $\mathcal{S}$-TERM($t_1$,$\theta$) in
  .4          let $s_2$ = $\mathcal{S}$-TERM($t_2$,$\theta$) in mk-PLUS($s_1$,$s_2$),
  .5        mk-MULT($t_1$,$t_2$) $\to$
  .6          let $s_1$ = $\mathcal{S}$-TERM($t_1$,$\theta$) in
  .7          let $s_2$ = $\mathcal{S}$-TERM($t_2$,$\theta$) in mk-MULT($s_1$,$s_2$),
  .8        mk-SIGN(t)       $\to$
  .9          let s = $\mathcal{S}$-TERM(t,$\theta$) in mk-SIGN(s)
38.10  type: $\mathcal{S}$-EXPR: EXPR $\times$ SUBSTITUTION $\to$ EXPR
```

Annotations to $\mathcal{S}$-*EXPR*:

.0 Substitute all terms in expressions.

End of annotations

Appendix E

Standard Predicates

E.1 Representation of numbers

We represent numbers as rational numbers.

```
1.    NUMBER        = $Q$
2.    $Q$           :: $(1 \mid -1) \times \mathsf{N}_0 \times \mathsf{N}_1$
```

Floating point division may introduce inaccuracy when rounding off. This is avoided in this representation. However, the accuracy has its price, because we need to compute the factorial numbers when adding two numbers and both the subtraction and the comparative functions are defined on addition. The implementation of all these functions can be found in section G.3.

E.2 Comparative Predicates

E.2.1 Less Than

```
1.    $less\text{-}than(t_1,t_2) \triangleq$
 .1     let mk-$CON(c_1) = t_1$, mk-$CON(c_2) = t_2$ in
 .2       cases $c_1$ :
 .3         mk-STRING$(s_1) \rightarrow$
 .4           let mk-STRING$(s_2) = c_2$ in
 .5             $s_1 < s_2$
 .6         T              $\rightarrow$
 .7           let mk-$Q(sign,,) = minus(c_1,c_2)$ in
 .8             $sign = -1$
1.9   type: $less\text{-}than : CON \times CON \times$ BOOL
```

Annotations to *less-than*:

.1 − .2 The function is overloaded to handle both strings and numbers.
.3 − .5 Strings are compared acording to the alfabetic order.
.6 − .8 Subtract the two numbers, t_1 is less than t_2 if the result is negative.

End of annotations

E.2.2 Less Than or Equal

```
2.    less-equal(t_1,t_2) ≜
 .1      let mk-CON(c_1) = t_1, mk-CON(c_2) = t_2 in
 .2         cases c_1 :
 .3            mk-STRING(s_1) →
 .4               let mk-STRING(s_2) = c_2 in
 .5                  s_1 ≤ s_2
 .6            T               →
 .7               let mk-Q(sign,numerator,) = minus(c_1,c_2) in
 .8                  (sign = -1) ∨ (numerator = 0)
2.9   type: less-equal : CON × CON × BOOL
```

Annotations to *less-equal*:
.1 − .2 The function is overloaded to handle both strings and numbers.
.3 − .5 Strings are compared acording to the alfabetic order.
.6 − .8 Subtract the two numbers, t_1 is less than or equal to t_2 if the result is negative or zero.

End of annotations

E.2.3 Not Equal

```
3.    not-equal(t_1,t_2) ≜
 .1      let mk-CON(c_1) = t_1, mk-CON(c_2) = t_2 in
 .2         cases c_1 :
 .3            mk-STRING(s_1) →
 .4               let mk-STRING(s_2) = c_2 in
 .5                  s_1 ≠ s_2
 .6            T               →
 .7               let mk-Q(,numerator,) = minus(c_1,c_2) in
 .8                  numerator ≠ 0
3.9   type: not-equal : CON × CON × BOOL
```

Annotations to *not-equal*:
.1 − .2 The function is overloaded to handle both strings and numbers.
.3 − .5 Strings are compared acording to the alfabetic order.

.6 − .8 Subtract the two numbers, t_1 is different from t_2 if the result is different from zero.

End of annotations

E.3 Arithmetical Predicates

E.3.1 Times

```
4.    $times(\underline{\textbf{mk-}}Q(s_1,n_1,d_1),\underline{\textbf{mk-}}Q(s_2,n_2,d_2)) \triangleq$
 .1     $\underline{\textbf{let}}\ s_3 = s_1 * s_2,$
 .2          $n_3 = n_1 * n_2,$
 .3          $d_3 = d_1 * d_2\ \underline{\textbf{in}}$
 .4        $\underline{\textbf{let}}\ (n_4,d_4) = reduce\text{-}fraction(n_3,d_3)\ \underline{\textbf{in}}$
 .5          $\underline{\textbf{mk-}}Q(s_3,n_4,d_4)$
4.6   $\underline{\textbf{type}}: times : CON \times CON \rightarrow CON$
```

Annotations to *times*:

.1 Multiply the signs, the numerators, and the denominator.

End of annotations

E.3.2 Divide

```
5.    $divide(t_1,\underline{\textbf{mk-}}Q(s_2,n_2,d_2)) \triangleq$
 .1     $\underline{\textbf{if}}\ n_2 = 0$
 .2     $\underline{\textbf{then}}\ \underline{\textbf{let}}\ \underline{\textbf{mk-}}Q(s_1,n_1,d_1) = t_1\ \underline{\textbf{in}}$
 .3              $\underline{\textbf{if}}\ n_1 = 0$
 .4              $\underline{\textbf{then}}$ ANONYMOUS VARIABLE
 .5              $\underline{\textbf{else}}\ \underline{\textbf{error}}$
 .6     $\underline{\textbf{else}}\ times(t_1,\underline{\textbf{mk-}}Q(s_2,d_2,n_2))$
5.7   $\underline{\textbf{type}}: divide : CON \times CON \rightarrow CON$
```

Annotations to *divide*:

.1 − .5 Division by zero is normaly not allowed but we will define zero divided by zero as valid for all numbers and return an anonymous variable.

.6 Otherwise division is defined from multiplication.

End of annotations

E.3.3 Plus

```
6.    plus(mk-Q(s_1,n_1,d_1),mk-Q(s_2,n_2,d_2)) ≜
 .1      let f_1 = factorize(d_1),
 .2          f_2 = factorize(d_2) in
 .3      let cd = union-factor(f_1,f_2) in
 .4      let rf_1 = reduce-factors(cd,f_1),
 .5          rf_2 = reduce-factors(cd,f_2) in
 .6      let r_1 = unfactorize(rf_1),
 .7          r_2 = unfactorize(rf_2),
 .8          rd = unfactorize(cd) in
 .9      let n_3 = s_1 * n_1 * r_1 + s_2 * n_2 * r_2 in
 .10     let s_3 = if n_3 ≥ 0 then 1 else -1 in
 .11     let n_4 = abs(n_3) in
 .12     let (n_5,d_5) = reduce-fraction(n_4,rd) in
 .13        mk-Q(s_3,n_5,d_5)
6.14  type: plus : CON × CON → CON
```

Annotations to *plus*:

.0 The auxiliary functions in this function are not specified but their implementation can be found in section G.3.

.1 The two rational numbers are added as one normaly add two fractions by finding the common denominator, and so on.

End of annotations

E.3.4 Minus

```
7.    minus(t_1,mk-Q(s_2,n_2,d_2)) ≜
 .1      plus(t_1,mk-Q(-s_2,d_2,n_2))
7.2   type: minus : CON × CON → CON
```

Annotations to *minus*:

.1 Subtraction is defined from addition.

End of annotations

E.3.5 Sign

```
8.    sign(mk-Q(s,n,d)) ≜
 .1      mk-Q(-s,d,n))
8.2   type: sign : CON → CON
```

Annotations to *sign*:

.1 Switch the sign.

End of annotations

Appendix F

The Parser Generator Input

The notation for the *parser generator input* is quite similar to the BNF description in section 2.2.9. E.g. the line

```
Datalog = Query Dbase -> datalog(Query,Dbase)
```

means that a `Datalog` program consists of (`=`) a `Query` and (`space`) a database (`Dbase`). We could not use the word “Database” because it is a reserved word in [PDC PROLOG]. When the parser recognizes this construction it generates (`->`) the functor `datalog(Query,Dbase)`. Alternatives are separated by a comma, see e.g. the three alternatives to `AdornBinding`.

```
productions
Datalog      = Query Dbase                           -> datalog(Query,Dbase)
Dbase        = Rule*
Query        = questionmark minus QLiteral punctum   -> query(QLiteral)
Rule         = Head colon minus Body punctum         -> rule(Head,Body),
               Head punctum                          -> rule2(Head),
               id(STRING)
                lshp int(INTEGER) rshp equal
                lshp Number semicolon
                Number rshp punctum                  -> range(STRING,INTEGER,Number,Number),
               id(STRING) colon int(INTEGER) punctum -> safety(STRING,INTEGER),
               id(STRING) ltub AdornList rtub punctum -> adornment(STRING,AdornList)

AdornList    = AdornBinding+ separator comma
AdornBinding = pre-bound                             -> bb,
               free-bound                            -> fb,
               post-free                             -> ff

Head         = id(STRING) lpar Termlist rpar         -> head(STRING,Termlist),
               id(STRING)                            -> head2(STRING)

Termlist     = Expr+ separator comma

Body         = Literal+ separator et
Literal      = Atom                                  -> pos(Atom),
               non lpar Atom rpar                    -> neg(Atom)

Atom         = id(STRING) lpar Termlist rpar         -> head(STRING,Termlist),
               id(STRING)                            -> head2(STRING),
               Expr less Expr                        -> less-than(Expr,Expr),
               Expr less-equal Expr                   -> less-equal(Expr,Expr),
               Expr greater-equal Expr                -> greater-equal(Expr,Expr),
               Expr greater Expr                      -> greater-than(Expr,Expr),
               Expr exclamationmark equal Expr        -> not-equal(Expr,Expr),
               Expr equal Expr                        -> equal(Expr,Expr)

Expr         = Expr plus Expr                         -> plus(Expr,Expr),
               Expr minus Expr                        -> minus(Expr,Expr)
               --
               Expr mult Expr                         -> mult(Expr,Expr),
               Expr div Expr                          -> div(Expr,Expr)
               --
               var(STRING)                            -> var(STRING),
               int(INTEGER)                           -> int(INTEGER),
               str(STRING)                            -> str(STRING),
               id(STRING)                             -> str(STRING),
               anonymousvariable                      -> anonymousvariable,
               minus Expr                             -> sign(Expr),
               lpar Expr rpar                         -> Expr

QLiteral     = QAtom                                  -> pos(QAtom)

QAtom        = id(STRING) lpar QTermlist rpar         -> head(STRING,QTermlist),
               id(STRING)                             -> head2(STRING)

QTermlist    = QExpr+ separator comma

QExpr        = var(STRING)                            -> var(STRING),
               minus int(INTEGER) div int(INTEGER)    -> signrat(INTEGER,INTEGER),
               int(INTEGER) div int(INTEGER)          -> rat(INTEGER,INTEGER),
               minus int(INTEGER)                     -> sign(INTEGER),
               int(INTEGER)                           -> int(INTEGER),
               str(STRING)                            -> str(STRING),
               anonymousvariable                      -> anonymousvariable,
               id(STRING)                             -> str(STRING)


Number       = minus int(INTEGER) div int(INTEGER)    -> signrat(INTEGER,INTEGER),
               int(INTEGER) div int(INTEGER)          -> rat(INTEGER,INTEGER),
               minus int(INTEGER)                     -> sign(INTEGER),
               int(INTEGER)                           -> int(INTEGER)
```

Comment: A 0-ary fact like `p.` is translated into `rule2(head2(p))` and to ease the overview we have to transform it into `rule(head(p,[ ]),[ ])`. The parser generator could not deal with it differently.

Appendix G

Interpreter Source Listing

This section contains the complete source text for the interpreter. The program is compiled as a *project* consisting of six *modules*. Each module has its own section with local *include* files. Shared include files are found in the section section (datalog.pro). The file globals.pro is included in front of all modules.

G.1 GLOBAL globals.pro

```
include "tdoms.pro"
include "color.pro"

include "datalog.dom"

global domains
  file = outfile

  CURSOR = INTEGER
  PROCID = STRING


global domains
  relation = r(STRING,Termlist,forest)
  forest = tree*
  tree = t(Expr,forest); leaf

  forests = forest*

  Program = Step*
  Step = init(STRING) ; step(Dbase) ; repeat(Dbase)

  state = relation*


  DependGraf = Depends*
  Depends = Depend*
  Depend = a(STRING,Strings)
  Strings = STRING*

  binding = b(STRING,STRING)
  bindings = binding*

   OccurSet = Occur*
   Occur    = occur(Expr,Integer)
   SubstSet = Subst*
   Subst    = sub(Expr,Expr)


strata  = strata(string,integer)
stratum = strata*

varset  = string*

arity   = arity(string,integer) % arity = -1 means conflict.
map = arity*

atype   = string() ; number()
arglist = atype*
ptype   = ptype(string,arglist)
ptypes  = ptype*


Code     = unfold ; intro_equal ; intro_fail
Codes    = Code*
RuleCode = rulecode(string,Codes)
ProgCode = RuleCode*


  adorn = adorn(STRING,forest)





% QSQR aux domain


Substitutions = Substitution*
Substitution = SubBinding*
SubBinding = binding(Expr,Expr) ; dummy


TSS = Termlist*

BL = FB*
FB = fri() ; bundet()
```

```
GLOBAL DATABASE
  range_db (STRING,INTEGER,Number,Number)
  safety_db (STRING,INTEGER)
  adornment_db (STRING,Adornlist)
  fact_edb (STRING,Termlist)
  query_db (STRING,Forest)
  rulegraph_db (Step)


%  global_state (state)

  source (STRING)
  filename (STRING)
  pdwstate (ROW,COL,SYMBOL,ROW,COL)
  insmode
  lineinpstate (STRING,COL)
  lineinpflag
  error (STRING,CURSOR)







global predicates
  interpret (Dbase,Query) - (i,i)
  interpret_program (Query,relation) - (i,o)

  load (STRING,state,relation) - (i,i,o)


  convert_rules (Dbase,Dbase) - (i,o)


% Write predicates
  vis_resultat2 (Dbase) - (i)
  vis_resultat2 (Depends) - (i)
  vis_resultat2 (DependGraf) - (i)
  vis_resultat2 (Codes) - (i)
  vis_resultat2 (TSS) - (i)
  vis_resultat2 (Substitution) - (i)
  vis_resultat2 (Substitutions) - (i)
  vis_resultat2 (Body) - (i)
  vis_resultat3 (Program) - (i)

  nondeterm vis_all_facts -
  nondeterm vis_all_range -
  nondeterm vis_all_safety -
  nondeterm vis_all_adornment -

  vis_number(Number) - (i)

vis_db (Dbase) - (i)
vis_rule (Rule) - (i)
vis_termlist (Termlist) - (i)
vis_termlist1 (Termlist) - (i)
vis_body (Body) - (i)
vis_body1 (Body) - (i)
vis_term (Expr) - (i)
vis_literal (Literal) - (i)
vis_atom (Atom) - (i)

vis_query (Query) - (i)
vis_qtermlist (QTermlist) - (i)
vis_qtermlist1 (QTermlist) - (i)
vis_qterm (QExpr) - (i)

vis_ass_term (Expr) - (i)


% Math predicates

  times (Expr,Expr,Expr) - (i,i,o)
  divide (Expr,Expr,Expr) - (i,i,o)
  plus (Expr,Expr,Expr) - (i,i,o)
  minus (Expr,Expr,Expr) - (i,i,o)

  less_than (Expr,Expr) - (i,i)
  less_equal (Expr,Expr) - (i,i)
  not_equal (Expr,Expr) - (i,i)
  equal (Expr,Expr) - (i,i)

  rational2real (Expr,Real) - (i,o)
  number2real (Number,Real) - (i,o)



CONSTANTS
  outname = "DATALOG.OUT"
  indent  = "\t"
```

G.1.1 INCLUDE tdoms.pro

```
/*****************************************************************
     Turbo Prolog Toolbox
     (C) Copyright 1987 Borland International.

 In order to use the tools, the following domain declarations
 should be included in the start of your program
*****************************************************************/

GLOBAL DOMAINS
  ROW, COL, LEN, ATTR   = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
      end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
      ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
      ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec
```

G.1.2 INCLUDE color.pro

```
constants /*mono/ color*/
  color_edit_t =       14
  color_edit_f =        9

  color_popup_t =       11
  color_popup_f =        9

  color_show_t = 3
  color_show_f =        9

  color_dir_t =       14
  color_dir_f =        9
```

G.1.3 INCLUDE datalog.dom

```
/************************************************************
DOMAIN DEFINITIONS
************************************************************/

GLOBAL DOMAINS  %FP 06.06.91
  Datalog         = datalog(Query,Dbase)

  Dbase           = Rule*

  Query           = query(QLiteral)

  Rule            = rule(Head,Body);
    rule2(Head);
    range(STRING,INTEGER,Number,Number);
    safety(STRING,INTEGER);
    adornment(STRING,AdornList)

  AdornList       = AdornBinding*

  AdornBinding    = bb();
    fb();
    ff()

  Head            = head(STRING,Termlist);
    head2(STRING)

  Termlist        = Expr*

  Body            = Literal*

  Literal         = pos(Atom);
    neg(Atom)

  Atom            = head(STRING,Termlist);
    head2(STRING);
    less_than(Expr,Expr);
    less_equal(Expr,Expr);
    greater_equal(Expr,Expr);
    greater_than(Expr,Expr);
    not_equal(Expr,Expr);
    equal(Expr,Expr)

  Expr            = plus(Expr,Expr);
    minus(Expr,Expr);
    mult(Expr,Expr);
    div(Expr,Expr);
    var(STRING);
    int(INTEGER);
    str(STRING);
    anonymousvariable();
    sign(Expr);
    q(INTEGER,INTEGER,INTEGER) % NEW TYPE INSERTED 9/8-91 FP

  QLiteral        = pos(QAtom)

  QAtom           = head(STRING,QTermlist);
    head2(STRING)

  QTermlist       = QExpr*

  QExpr           = var(STRING);
    signrat(INTEGER,INTEGER);
    rat(INTEGER,INTEGER);
    sign(INTEGER);
    int(INTEGER);
```

```
    anonymousvariable();
    str(STRING)

  Number          = signrat(INTEGER,INTEGER);
    rat(INTEGER,INTEGER);
    sign(INTEGER);
    int(INTEGER)


  TOK   = comma();
    et();
    questionmark();
    minus();
    punctum();
    collon();
    id(STRING);
    lshp();
    int(INTEGER);
    rshp();
    equal();
    semicollon();
    ltub();
    rtub();
    pre_bound();
    free_bound();
    post_free();
    lpar();
    rpar();
    non();
    less();
    less_equal();
    greater_equal();
    greater();
    exclamationmark();
    plus();
    mult();
    div();
    var(STRING);
    str(STRING);
    anonymousvariable();
    nill
```

G.2 MODULE datalog.pro

```
project "datalog"
include "globals.pro"
include "settools.pro"
include "iswf-DAT.pro"
include "tpreds.pro"
include "status.pro"
include "pulldown.pro"
include "lineinp.pro"
include "filename.pro"


PREDICATES
  scan_error (STRING,CURSOR)
  change (DBASEDOM)
  get_tekst
  finn(INTEGER)
  main

include "datalog.sca"

CLAUSES
%  change(global_state(_)):- retract(global_state(_)),fail.
  change(source(_)) :- retract(source(_)),fail.
  change(filename(_)) :- retract(filename(_)),fail.
  change(X) :- assert(X).


  get_tekst
     :- comline(Comline),
        existfile(Comline),!,
        file_str(Comline,Tekst),!,
        change(filename(Comline)),
        change(source(Tekst)),
        unreadchar('e'), !.
  get_tekst
     :- change(source("")),
        unreadchar('l'),
        unreadchar('f'), !.




main :- trace(off),

/*
          1         2         3         4         5         6         7
01234567890123456789012345678901234567890123456789012345678901234567890
       COMPILE            EDIT              FILES               QUIT
*/
  textmode(R,C), R1 = R-4,
  makewindow(1,color_show_t,color_show_f,"Editor",3,0,R1,C),
  makestatus(112," Select with arrows or use first upper case letter"),
%  disk(OLDPATH),
%  frontchar(OLDPATH,DRIVE,_),
%  frontchar(NEWPATH,DRIVE,":\\DEMO"),

%  write(OLDPATH," ",NEWPATH),nl, readchar(_),

%  newdisk(NEWPATH),!,
  get_tekst,

  pulldown(color_popup_t,
  [ curtain(7,"Interpreter",[]),
    curtain(26,"Editor",[]),
    curtain(44,"Files",["Load","Save","Directory"]),
            curtain(64,"Quit" ,[])
          ],_,_),
  removewindow,
  newdisk(OLDPATH).


finn(I) :- str_int(S,I), concat("Error: ",S,T), display(T).

%GOAL trap(main,ExitCode,finn(ExitCode)).
goal main.

PREDICATES
  strlist_str(STRINGLIST,STRING)
  ed(STRING,CURSOR)
  resetflags
  nondeterm repparse
  parse
  better_error(CURSOR)
  new_error(STRING,CURSOR)


PREDICATES
  expect(CURSORTOK,TOKL,TOKL)
  syntax_error(STRING,TOKL)
  checkempty(TOKL)

  adjust_rules (Dbase,Dbase)
  save_user_info(Rule)

  adjust_head (Head,Head)
  adjust_body (Body,Body)
  adjust_atom (Atom,Atom)

  adjust_query (Query,Query)
  adjust_qatom (QAtom,QAtom)
```

```
include "datalog.par"

CLAUSES
  better_error(CURSOR):-
error(_,OLDCURSOR),OLDCURSOR>=CURSOR,!,fail.
  better_error(_).

  new_error(_,_):-retract(error(_,_)),fail.
  new_error(MSG,CURSOR):-assert(error(MSG,CURSOR)).

  expect(TOK,[TOK|L],L):-!.
  expect(t(TOK,_),[t(_,CURSOR)|_],_):-
better_error(CURSOR),
str_tok(STR,TOK),
concat(STR," expected",MSG),
new_error(MSG,CURSOR),fail.

  syntax_error(PROD,[t(_,CURSOR)|_]):-
better_error(CURSOR),
concat("Syntax error in ",PROD,MSG),
new_error(MSG,CURSOR),fail.

  scan_error(MSG,CURSOR):-ed(MSG,CURSOR),fail.

  checkempty([]):-!.
  checkempty([t(_,CURSOR)|_]):-
better_error(CURSOR),
new_error("Syntax error",CURSOR).

  strlist_str([],"").
  strlist_str([H|T],STR):-
concat(H," ",H1),
strlist_str(T,STR1),
concat(H1,STR1,STR).

  ed(MSG,CURSOR):-
source(TXT),
edit(TXT,TXT1,"","",MSG,CURSOR,"",RET),
RET><1,!,
change(source(TXT1)).

  resetflags:-retract(error(_,_)),fail.
  resetflags.

  repparse.
  repparse:-error(MSG,CURSOR),ed(MSG,CURSOR),!,repparse.


  parse:-
repparse,
resetflags,
source(STR1),
tokl(0,STR1,L),

/* Empty all information databases. */
retractall(range_db(_,_,_,_)),
   retractall(safety_db(_,_)),
   retractall(adornment_db(_,_)),
   retractall(fact_edb(_,_)),
   retractall(rulegraph_db(_)),
retractall(query_db(_,_)),



s_datalog(L,L1,datalog(Query2,DB2)),
list2set            (DB2,DB3),
adjust_rules        (DB3,DB4),
adjust_query        (Query2,Query),

convert_rules       (DB4,DB),


iswf_DATALOG(Query,DB),

checkempty(L1),!,
textmode(R,C),
makewindow(2,color_show_t,color_show_f,"",0,0,R,C),

/*----------------------------------------------*/
%FP
  interpret(DB,Query),

/*----------------------------------------------*/
removewindow.


  parse:-
write(">> Parsing aborted"),nl,beep.

  pdwaction(1,0):-
shiftwindow(OLD),shiftwindow(1),
parse,
shiftwindow(OLD),
refreshstatus.
  pdwaction(2,0):-
shiftwindow(OLD),shiftwindow(1),
source(TXT),
filename(Filename),                             %FP060291
edit(TXT,TXT1,"",Filename,"Press F10 to return to menu",0,
"DATALOG.HLP",RET),
shiftwindow(OLD),
RET><1,!,
change(source(TXT1)),
refreshstatus.
  pdwaction(2,0):-refreshstatus.
  pdwaction(3,1):-
readfilename(5,40,color_edit_t,color_edit_f,txt,"",NEW),
change(filename(NEW)),
file_str(NEW,NEWSOURCE),!,
change(source(NEWSOURCE)),
shiftwindow(OLDW),
shiftwindow(1),
window_str(NEWSOURCE),
shiftwindow(OLDW),
refreshstatus.
  pdwaction(3,1).
  pdwaction(3,2):-
source(SOURCE),
filename(OLD),
readfilename(5,30,color_edit_t,color_edit_f,txt,OLD,NEW),
change(filename(NEW)),
file_str(NEW,SOURCE),
refreshstatus,!.
  pdwaction(3,2).
  pdwaction(3,3):-setdir(5,40,color_dir_t,color_dir_f).
  pdwaction(4,0):- fail.

%  global_state([]).
  filename("datalog.txt").
  source("").

adjust_rules([],[]) :- !.
adjust_rules([rule2(H2)|R2],[rule(H,[])|R])
   :- adjust_head(H2,H),
      adjust_rules(R2,R),!.
adjust_rules([rule(H2,B2)|R2],[rule(H,B)|R])
   :- adjust_head(H2,H),
      list2set(B2,B3),
      adjust_body(B3,B),
      adjust_rules(R2,R),!.
adjust_rules([Otherwise|R2],R)
   :- save_user_info(Otherwise),
      adjust_rules(R2,R),!.


save_user_info(range(ID,Arg,Low,Upp))
   :- assert(range_db(ID,Arg,Low,Upp)),!.
save_user_info(safety(ID,Level))
   :- assert(safety_db(ID,Level)),!.
save_user_info(adornment(ID,Adornlist))
   :- assert(adornment_db(ID,Adornlist)),!.
```

```
adjust_head(head2(S),head(S,[])) :- !.
adjust_head(Head,Head).

adjust_body([],[]) :- !.
adjust_body([pos(Atom2)|Rest2],[pos(Atom)|Rest])
   :- adjust_atom(Atom2,Atom),
      adjust_body(Rest2,Rest), !.
adjust_body([neg(Atom2)|Rest2],[neg(Atom)|Rest])
   :- adjust_atom(Atom2,Atom),
      adjust_body(Rest2,Rest), !.

adjust_atom(head2(S),head(S,[])) :- !.
adjust_atom(Atom,Atom).


adjust_query(query(pos(QAtom2)),query(pos(QAtom)))
   :- adjust_qatom(QAtom2,QAtom), !.

adjust_qatom(head2(S),head(S,[])) :- !.
adjust_qatom(QAtom,QAtom).
```

G.2.1 INCLUDE settools.pro

```
predicates
% insert (element,set,set)
  insert (Rule,Dbase,Dbase)
  insert (Literal,Body,Body)
  insert (strata,stratum,stratum)
  insert (arity,map,map)
  insert (depend,depends,depends)
  insert (Code,Codes,Codes)
  insert (string,strings,strings)
  insert (Subst,SubstSet,SubstSet)
  insert (Termlist,TSS,TSS)
  insert (Substitution,Substitutions,Substitutions)
  insert (Expr,Termlist,Termlist)


% nondeterm member (element,set)
  nondeterm member (Rule,Dbase)
  nondeterm member (Literal,Body)
  nondeterm member (strata,stratum)
  nondeterm member (arity,map)
  nondeterm member (depend,depends)
  nondeterm member (Code,Codes)
  nondeterm member (string,strings)
  nondeterm member (Occur,OccurSet)
  nondeterm member (Subst,SubstSet)
  nondeterm member (SubBinding,Substitution)
  nondeterm member (Expr,Termlist)
  nondeterm member (Substitution,Substitutions)
  nondeterm member (Termlist,TSS)

% remove (element,set,set)
  remove (Rule,Dbase,Dbase)
  remove (Literal,Body,Body)
  remove (strata,stratum,stratum)
  remove (arity,map,map)
  remove (depend,depends,depends)
  remove (Code,Codes,Codes)
  remove (string,strings,strings)
  remove (Occur,OccurSet,OccurSet)
  remove (Subst,SubstSet,SubstSet)
  remove (Substitution,Substitutions,Substitutions)
  remove (SubBinding,Substitution,Substitution)

% union (set,set,set)
  union (Dbase,Dbase,Dbase)
  union (Body,Body,Body)
  union (Depends,Depends,Depends)
  union (Strings,Strings,Strings)
  union (Codes,Codes,Codes)
  union (SubstSet,SubstSet,SubstSet)

% difference (set,set,set)
  difference (Dbase,Dbase,Dbase)
  difference (Body,Body,Body)

% sub_set (set,set)
  sub_set (Dbase,Dbase)
  sub_set (Body,Body)

% equal_set (set,set)
  equal_set (Body,Body)

% append (list,list,list)
  append (Dbase,Dbase,Dbase)
  append (Body,Body,Body)
  append (Termlist,Termlist,Termlist)
  append (OccurSet,OccurSet,OccurSet)
  append (Forest,Forest,Forest)


  append (DependGraf,DependGraf,DependGraf)
  append (Depends,Depends,Depends)
  append (Program,Program,Program)
  append (Substitution,Substitution,Substitution)

% list2set (list,set)
  list2set (Dbase,Dbase)
  list2set (Body,Body)


% card (set,integer)
  card (stratum,integer)
  card (Termlist,integer)
  card (QTermlist,integer)
  card (Depends,integer)



clauses



% insert : TYPE x TYPE-set -> TYPE-set

  insert (T,[],[T]) :- !.
  insert (T,[T|R],[T|R]) :- !.
  insert (T,[A|R],[A|B]) :- insert(T,R,B).



% member : TYPE x TYPE-set ->

  member(X,[X|_]).
  member(X,[_|R])  :- member(X,R).



% remove : TYPE x TYPE-set -> TYPE-set

  remove (_,[],[]) :- !.
  remove (T,[T|R],R) :- !.
  remove (T,[A|R],[A|B]) :- remove(T,R,B).


% union : TYPE-set x TYPE-set -> TYPE-set

  union ([],B,B) :- !.
  union (B,[],B) :- !.
  union (A,B,C) :- member(T,B), !,
      remove(T,B,S1),
      union(A,S1,S2),
      insert(T,S2,C).

% difference : TYPE-set x TYPE-set -> TYPE-set

  difference ([],_,[]) :- !.
  difference (A,[],A) :- !.
  difference (A,B,C) :- member(T,B), !,
      remove(T,B,S1),
      remove(T,A,S2),
      difference(S2,S1,C).


% sub_set : TYPE-set x TYPE-set -> BOOL

  sub_set ([E|R],S)             :- member(E,S), !,
                                   sub_set(R,S), !.
  sub_set ([],_).

% equal_set : TYPE-set x TYPE-set -> BOOL

  equal_set ([E|R],S1) :- member(E,S1), !,
      remove(E,S1,S2),
      equal_set(R,S2), !.
  equal_set ([],[]).

% append : TYPE* x TYPE* -> TYPE*

  append([],L,L).
  append([X|L1],L2,[X|L3]):- append(L1,L2,L3).


% list2set : TYPE* -> TYPE-set


  list2set ([],[])              :- !.
  list2set ([E|R],S)            :- list2set(R,S1),
                                   insert(E,S1,S).

% card : TYPE-set -> INTG

  card([],0)                    :- !.
  card([_|R],C1)   :- card(R,C),!, C1 = C+1.
```

G.2.2 INCLUDE iswf-dat.pro

```
/* domains.pro
DOMAINS

strata  = strata(string,integer)
stratum = strata*

varset  = string*

arity   = arity(string,integer) % arity = -1 means conflict.
map = arity*
```

```
atype   = string() ; number()
arglist = atype*
ptype   = ptype(string,arglist)
ptypes  = ptype*
*/

PREDICATES

iswf_DATALOG(Query,Dbase)

stratified(Dbase)
   remove_range(Dbase,Dbase)
   remove_rule(Dbase,Dbase)
   init_stratum(Dbase,Stratum)
   strata(Dbase,Stratum,Stratum,integer,integer)
      strata1(Dbase,Stratum,Stratum)
      strata2(Body,Stratum,integer,integer)
      get_strata(string,Stratum,integer)
      maximum(integer,integer,integer)
      maxrng(Stratum,integer)

aritycheck(Query,Dbase,map)
   arityconflict(map)
      caseconflict(stringlist)
      writeset(stringlist)
   aritydb(Dbase,map,map)
   arityrule(Rule,map,map)
   aritybody(Body,map,map)
   arityliteral(Literal,map,map)
   arityatom(Atom,map,map)


rangecheck(Dbase,map,integer)
   test_range(string,integer,integer,integer)
   compare_number(Number,Number)
   convert_number(Number,Real)



typecheck(Query,Dbase)
failcheck(Dbase)


CLAUSES

iswf_DATALOG(Query,Dbase1)
   :- file_str(outname,""),
      openwrite(outfile,outname),
      writedevice(outfile),

      remove_range(Dbase1,Dbase),
      stratified(Dbase),

      aritycheck(Query,Dbase,Map),
      remove_rule(Dbase1,Dbase2),
trace(on),
      rangecheck(Dbase2,Map,0),


      typecheck(Query,Dbase),
      failcheck(Dbase),

      closefile(outfile), !.

iswf_DATALOG(_,_)
   :- write("ERROR in iswf_DATALOG"), nl,
      closefile(outfile).



stratified(DB)
   :- init_stratum(DB,Stratum),
      card(Stratum,NoPred),
      strata(DB,Stratum,[],NoPred,MaxStrata),
      MaxStrata <= NoPred, write("The database is stratified."), nl, !.
stratified(_)
   :- write("FATAL ERROR: The database is not stratified."), nl.

   remove_range([rule(Head,Body)|Dbase1],[rule(Head,Body)|Dbase2])
      :- remove_range(Dbase1,Dbase2), !.
   remove_range([_|Dbase1],Dbase2)
      :- remove_range(Dbase1,Dbase2), !.
   remove_range([],[]).

   remove_rule([range(P,A,L,U)|Dbase1],[range(P,A,L,U)|Dbase2])
      :- remove_rule(Dbase1,Dbase2), !.
   remove_rule([_|Dbase1],Dbase2)
      :- remove_rule(Dbase1,Dbase2), !.
   remove_rule([],[]).

   init_stratum([rule(head(P,_),_)|Dbase],Stratum2)
      :- init_stratum(Dbase,Stratum1),
         insert(strata(P,1),Stratum1,Stratum2), !.
   init_stratum([],[]).

   strata(_,Stratum1,_,NP,Max)
      :- maxrng(Stratum1,Max),
         Max > NP, !.
   strata(_,Stratum1,Stratum1,_,Max)
      :- maxrng(Stratum1,Max), !.
   strata(DB,Stratum1,_,NP,Max)
      :- strata1(DB,Stratum1,Stratum3),
         strata(DB,Stratum3,Stratum1,NP,Max), !.

      strata1([rule(head(P,_),Body)|Dbase],Stratum1,Stratum4)
         :- get_strata(P,Stratum1,Min),!,
            strata2(Body,Stratum1,Min,Max),
            member(strata(P,Strata),Stratum1),!,
            remove(strata(P,Strata),Stratum1,Stratum2),
            insert(strata(P,Max),Stratum2,Stratum3),
            strata1(Dbase,Stratum3,Stratum4), !.
      strata1([],Stratum,Stratum).

      strata2([pos(head(P,_))|Body],Stratum,Strata1,Max)
         :- get_strata(P,Stratum,Strata2),
            maximum(Strata1,Strata2,Strata3),
            strata2(Body,Stratum,Strata3,Max),!.

      strata2([neg(head(P,_))|Body],Stratum,Strata1,Max)
         :- get_strata(P,Stratum,Strata2),
            Strata3 = Strata2 +1,
            maximum(Strata1,Strata3,Strata4),
            strata2(Body,Stratum,Strata4,Max),!.

      strata2([_|Body],Stratum,Strata,Max)
         :- strata2(Body,Stratum,Strata,Max),!.

      strata2([],_,Max,Max).



      get_strata(P,Stratum,Strata)
         :- member(strata(P,Strata),Stratum),!.
      get_strata(_,_,0).



      maximum(I1,I2,I1) :- I1 > I2, !.
      maximum(_,I2,I2).



      maxrng([],0) :- !.
      maxrng([strata(_,S1)|Stratum],Max)
         :- maxrng(Stratum,S2),!,
            maximum(S1,S2,Max),!.


aritycheck(query(pos(head(Q,TS))),DB,Map)
   :- aritydb(DB,[],Map),
      arityconflict(Map),
      card(TS,Arity),
      member(arity(Q,Arity),Map), !.
```

```
aritycheck(_,_,[])
   :- write("FATAL ERROR: Arity of query
             conflicts with the database."), nl.

   arityconflict(Map)
      :- findall(P,member(arity(P,-1),Map),PredConf),
         caseconflict(PredConf).

      caseconflict([]) :- write("Arity check done."), nl, !.
      caseconflict(PC)
         :- write("FATAL ERROR: The following predicates
                   conflict in arity: "),
            writeset(PC), nl, !.


      writeset([])    :- !.
      writeset([H])   :- write(H,"."), !.
      writeset([H|R]) :- write(H,", "), writeset(R).


   aritydb([],Map,Map) :- !.
   aritydb([Rule|DB],Map1,Map3)
      :- aritydb(DB,Map1,Map2), !,
         arityrule(Rule,Map2,Map3), !.

   arityrule(rule(head(P,TS),Body),Map1,Map2)
      :- aritybody([pos(head(P,TS))|Body],Map1,Map2), !.

   aritybody([],Map,Map) :- !.
   aritybody([Literal|Body],Map1,Map3)
      :- arityliteral(Literal,Map1,Map2), !,
         aritybody(Body,Map2,Map3), !.


   arityliteral(pos(Atom),Map1,Map2)
      :- arityatom(Atom,Map1,Map2), !.
   arityliteral(neg(Atom),Map1,Map2)
      :- arityatom(Atom,Map1,Map2), !.


   arityatom(head(P,TS),Map,Map)
      :- card(TS,Arity),
         member(arity(P,Arity),Map),!.

   arityatom(head(P,TS),Map1,Map3)
      :- member(arity(P,Arity2),Map1),!,
         card(TS,Arity1),
         not (Arity1 = Arity2),
         remove(arity(P,Arity2),Map1,Map2),
         insert(arity(P,-1),Map2,Map3), !.
   arityatom(head(P,TS),Map1,Map2)
      :- card(TS,Arity),
         insert(arity(P,Arity),Map1,Map2), !.

   arityatom(_,Map,Map).



rangecheck(_,[],_) :- !.
rangecheck([],_,0) :- write("Range check done."), nl, !.
rangecheck([],_,_) :- nl, !.
rangecheck([range(P,_,Lower,Upper)|Dbase],Map,Error)
   :- compare_number(Lower,Upper), !,
      write("FATAL ERROR: Interval error in range
             definition of predicate: ",P),nl,
      NewError = 1 + Error,
      rangecheck(Dbase,Map,NewError), !.
rangecheck([range(P,Arg,_,_)|Dbase],Map,Error)
   :- member(arity(P,Arity),Map), !,
      test_range(P,Arg,Arity,New), NewError = New + Error,
      rangecheck(Dbase,Map,NewError), !.
rangecheck([range(P,_,_,_)|Dbase],Map,Error)
   :- write("FATAL ERROR: Range definde for non
             existing predicate: ",P),nl,
      NewError = 1 + Error,
      rangecheck(Dbase,Map,NewError), !.

   test_range(_,Arg,Arity,0)
      :- Arity >= Arg, Arg > 0, !.
   test_range(P,_,-1,1) :- write("WARNING: No range check done
                                 on predicate: ",P), nl, !.
   test_range(P,_,0,1)
      :- write("FATAL ERROR: Range can not be defined for 0-ary
                predicate: ",P),nl, !.
   test_range(P,Arg,Arity,1)
      :- write("FATAL ERROR: Range argument outside
                predicate arity: ",
               P,"(1:",Arity,") - ",P,"[",Arg,"]"), nl, !.

   compare_number(Lower,Upper)
      :- convert_number(Lower,L),
         convert_number(Upper,U),
         L > U.

   convert_number(signrat(T,N),Num) :- Num = -T/N, !.
   convert_number(rat(T,N),Num) :- Num = T/N, !.
   convert_number(sign(T),Num) :- Num = -T, !.
   convert_number(int(T),Num) :- Num = T, !.
```

```
typecheck(_,_)  :- write("WARNING: No Type check done."), nl.



failcheck([rule(head(P,_),_)|Dbase])
   :- not (P = "fail"), failcheck(Dbase), !.
failcheck([range(P,_,_,_)|Dbase])
   :- not (P = "fail"), failcheck(Dbase), !.
failcheck([])
   :- write("Fail undefined."), nl,!.

failcheck(_) :- write("FATAL ERROR: Standard predicate fail
                       defined by the user."), nl.
```

G.2.3 INCLUDE tpreds.pro

```
/*****************************************************************
     Turbo Prolog Toolbox
     (C) Copyright 1987 Borland International.

 This module includes some routines which are used in nearly
 all menu and screen tools.
*****************************************************************/

/******************************************************************/
/* repeat */
/******************************************************************/

PREDICATES
  nondeterm repeat

CLAUSES
  repeat.
  repeat:-repeat.


/******************************************************************/
/* miscellaneous */
/******************************************************************/

PREDICATES
  maxlen(STRINGLIST,COL,COL)
   /* The length of the longest string */
  listlen(STRINGLIST,ROW)
   /* The length of a list     */
  writelist(ROW,COL,STRINGLIST)
   /* used in the menu predicates     */
  reverseattr(ATTR,ATTR)
   /* Returns the reversed attribute   */
  min(ROW,ROW,ROW)
  min(COL,COL,COL)
  min(LEN,LEN,LEN)
  min(INTEGER,INTEGER,INTEGER)
  max(ROW,ROW,ROW)
  max(COL,COL,COL)
  max(LEN,LEN,LEN)
  max(INTEGER,INTEGER,INTEGER)

CLAUSES
  maxlen([H|T],MAX,MAX1) :-
str_len(H,LENGTH),
LENGTH>MAX,!,
maxlen(T,LENGTH,MAX1).
  maxlen([_|T],MAX,MAX1) :- maxlen(T,MAX,MAX1).
  maxlen([],LENGTH,LENGTH).

  listlen([],0).
  listlen([_|T],N):-
listlen(T,X),
N=X+1.

  writelist(_,_,[]).
  writelist(LI,ANTKOL,[H|T]):-
field_str(LI,0,ANTKOL,H),
LI1=LI+1,
writelist(LI1,ANTKOL,T).

  min(X,Y,X):-X<=Y,!.
  min(_,X,X).

  max(X,Y,X):-X>=Y,!.
  max(_,X,X).

  reverseattr(A1,A2):-
bitand(A1,$07,H11),
bitleft(H11,4,H12),
bitand(A1,$70,H21),
bitright(H21,4,H22),
bitand(A1,$08,H31),
A2=H12+H22+H31.


/****************************************************************/
/* Find letter selection in a list of strings */
/*      Look initially for first uppercase letter. */
/*      Then try with first letter of each string. */
/****************************************************************/

PREDICATES
  upc(CHAR,CHAR)  lowc(CHAR,CHAR)
  try_upper(CHAR,STRING)
  tryfirstupper(CHAR,STRINGLIST,ROW,ROW)
  tryfirstletter(CHAR,STRINGLIST,ROW,ROW)
  tryletter(CHAR,STRINGLIST,ROW)

CLAUSES
  upc(CHAR,CH):-
CHAR>='a',CHAR<='z',!,
char_int(CHAR,CI), CI1=CI-32, char_int(CH,CI1).
  upc(CH,CH).

  lowc(CHAR,CH):-
CHAR>='A',CHAR<='Z',!,
char_int(CHAR,CI), CI1=CI+32, char_int(CH,CI1).
  lowc(CH,CH).

  try_upper(CHAR,STRING):-
frontchar(STRING,CH,_),
CH>='A',CH<='Z',!,
CH=CHAR.
  try_upper(CHAR,STRING):-
frontchar(STRING,_,REST),
try_upper(CHAR,REST).

  tryfirstupper(CHAR,[W|_],N,N) :-
try_upper(CHAR,W),!.
  tryfirstupper(CHAR,[_|T],N1,N2) :-
N3 = N1+1,
tryfirstupper(CHAR,T,N3,N2).

  tryfirstletter(CHAR,[W|_],N,N) :-
frontchar(W,CHAR,_),!.
  tryfirstletter(CHAR,[_|T],N1,N2) :-
N3 = N1+1,
tryfirstletter(CHAR,T,N3,N2).

  tryletter(CHAR,LIST,SELECTION):-
upc(CHAR,CH),tryfirstupper(CH,LIST,0,SELECTION),!.
  tryletter(CHAR,LIST,SELECTION):-
lowc(CHAR,CH),tryfirstletter(CH,LIST,0,SELECTION).



/*****************************************************************/
/* adjustwindow takes a windowstart and a windowsize and adjusts */
/* the windowstart so the window can be placed on the screen.  */
/* adjframe looks at the frameattribute: if it is different from */
/* zero, two is added to the size of the window  */
/****************************************************************/

PREDICATES
  adjustwindow(ROW,COL,ROW,COL,ROW,COL)
  adjframe(ATTR,ROW,COL,ROW,COL)

CLAUSES
  adjustwindow(LI,KOL,DLI,DKOL,ALI,AKOL):-
LI<25-DLI,KOL<80-DKOL,!,ALI=LI,AKOL=KOL.
  adjustwindow(LI,_,DLI,DKOL,ALI,AKOL):-
LI<25-DLI,!,ALI=LI,AKOL=80-DKOL.
  adjustwindow(_,KOL,DLI,DKOL,ALI,AKOL):-
KOL<80-DKOL,!,ALI=25-DLI, AKOL=KOL.
  adjustwindow(_,_,DLI,DKOL,ALI,AKOL):-
ALI=25-DLI, AKOL=80-DKOL.

  adjframe(0,R,C,R,C):-!.
  adjframe(_,R1,C1,R2,C2):-R2=R1+2, C2=C1+2.


/****************************************************************/
/*  Readkey */
/* Returns a symbolic key from the KEY domain         */
/****************************************************************/

PREDICATES
  readkey(KEY)
  readkey1(KEY,CHAR,INTEGER)
  readkey2(KEY,INTEGER)

CLAUSES
  readkey(KEY):-readchar(T),char_int(T,VAL),readkey1(KEY,T,VAL).

  readkey1(KEY,_,0):-!,readchar(T),char_int(T,VAL),readkey2(KEY,VAL).
  readkey1(cr,_,13):-!.
  readkey1(esc,_,27):-!.
  readkey1(break,_,3):-!.
  readkey1(tab,_,9):-!.
  readkey1(bdel,_,8):-!.
  readkey1(ctrlbdel,_,127):-!.
  readkey1(char(T),T,_) .

  readkey2(btab,15):-!.
  readkey2(del,83):-!.
  readkey2(ins,82):-!.
  readkey2(up,72):-!.
  readkey2(down,80):-!.
  readkey2(left,75):-!.
  readkey2(right,77):-!.
  readkey2(pgup,73):-!.
  readkey2(pgdn,81):-!.
  readkey2(end,79):-!.
  readkey2(home,71):-!.
  readkey2(ctrlleft,115):-!.
  readkey2(ctrlright,116):-!.
  readkey2(ctrlend,117):-!.
  readkey2(ctrlpgdn,118):-!.
  readkey2(ctrlhome,119):-!.
  readkey2(ctrlpgup,132):-!.
  readkey2(fkey(N),VAL):- VAL>58, VAL<70, N=VAL-58, !.
  readkey2(fkey(N),VAL):- VAL>=84, VAL<104, N=11+VAL-84, !.
  readkey2(otherspec,_).
```

G.2.4 INCLUDE status.pro

```
 /**********************************************************************

     Turbo Prolog Toolbox
     (C) Copyright 1987 Borland International.

           Status Line
            Uses window number 83 for a status line.
 **********************************************************************/

PREDICATES
  makestatus(ATTR,STRING)
  nondeterm tempstatus(ATTR,STRING)
  removestatus
  refreshstatus
  changestatus(STRING)

CLAUSES
  makestatus(ATTR,TXT):-
shiftwindow(OLD),
textmode(R,C), R1 = R-1,
makewindow(83,ATTR,0,"",R1,0,1,C),

field_str(0,0,C,TXT),
shiftwindow(OLD).

  tempstatus(ATTR,TXT):-makestatus(ATTR,TXT).
  tempstatus(_,_):-removestatus,fail.

  removestatus:-
shiftwindow(OLD),
shiftwindow(83), removewindow,
shiftwindow(OLD).

  refreshstatus:-
shiftwindow(OLD),
shiftwindow(83),
shiftwindow(OLD).

  changestatus(TXT):-
shiftwindow(OLD),
shiftwindow(83),
field_str(0,0,80,TXT),
shiftwindow(OLD).
```

G.2.5 INCLUDE pulldown.pro

```
/*****************************************************************

     Turbo Prolog Toolbox
     (C) Copyright 1987 Borland International.

PULL DOWN MENU

  This module implements a pulldown menu.

  The parameters are:
pulldown(ATTRIBUTE,MENULIST,CHOICE,SUBCHOICE)

  where
Attribute is used in all the windows
Menulist is the text for the menus
CHOICE is the selection from the horizontal menu
SUBCHOICE is the selection from the vertical menu
       (or zero if there is no vertical menu for
        the CHOICE horizontal item)
*****************************************************************/

/* ----- Include this database in your program ----
DATABASE
pdwstate(ROW,COL,SYMBOL,ROW,COL)

include tooldom and toolpred

And give some clauses for the pdwaction predicate

*/


DOMAINS
 MENUELEM=  curtain(COL,STRING,STRINGLIST)
 MENULIST=  MENUELEM*
 STOP   =  stop(); cont()


PREDICATES
  pulldown(ATTR,MENULIST,INTEGER,INTEGER)
  pdwaction(INTEGER,INTEGER)

  pdwkeyact(KEY,ROW,COL,SYMBOL,ROW,COL,COL,ATTR,MENULIST,STOP)
  pdwmovevert(COL,COL,ATTR,MENULIST)
  pdwindex(COL,MENULIST,MENUELEM)
  pdwindex(ROW,STRINGLIST,STRING)
  makepdwwindow1(ROW,COL,ROW,COL,ATTR,STRINGLIST,ROW)
  makepdwwindow(COL,ATTR,MENULIST,ROW,COL,ROW)
  writelistp(ROW,COL,ATTR,STRINGLIST)
  line_ver(ROW,ROW,COL)
  line_hor(COL,COL,ROW)
  lcorn(COL,CHAR)
  rcorn(COL,CHAR)
  pdwlistlen(MENULIST,COL)
  writepdwlist(ATTR,MENULIST)
  changepdwstate(DBASEDOM)
  check_removewindow(ROW)
  is_up(SYMBOL,ROW)
  nextcol(COL,COL,COL,COL)
  intense(ATTR,ATTR)
  intensefirstupper(ROW,COL,ATTR,STRING)
  intenseletter(ROW,COL,ATTR,STRING)
  pdwlist_strlist(MENULIST,STRINGLIST)

CLAUSES

/* draw pulldown window */
  line_ver(R1,R2,C):-
R2>R1,!, R=R1+1,
scr_char(R1,C,'I'),
line_ver(R,R2,C).
  line_ver(_,_,_).

  line_hor(C1,C2,R):-
C2>C1,!, C=C1+1,
scr_char(R,C1,'-'),
line_hor(C,C2,R).
  line_hor(_,_,_).

/* Make the pulldown window */
  makepdwwindow(NO,ATTR,MENULIST,LISTLEN,MAXLEN,FIRSTROW):-
pdwindex(NO,MENULIST,curtain(CCOL,_,LIST)),COL=CCOL,
ROW=2,
listlen(LIST,LISTLEN1),LISTLEN=LISTLEN1,
maxlen(LIST,0,MAXLEN),
makepdwwindow1(ROW,COL,LISTLEN,MAXLEN,ATTR,LIST,FIRSTROW).

/*  makepdwwindow1(_,_,_,_,_,_,0):-keypressed,!. */
  makepdwwindow1(_,_,0,_,_,_,0):-!.
  makepdwwindow1(ROW,COL,LISTLEN,MAXLEN,ATTR,LIST,1):-
NOOFROWS=LISTLEN+2, NOOFCOLS=MAXLEN+2,
adjustwindow(ROW,COL,NOOFROWS,NOOFCOLS,AROW,ACOL),
makewindow(81,ATTR,0,"",AROW,ACOL,NOOFROWS,NOOFCOLS),
writelistp(1,MAXLEN,ATTR,LIST),
cursor(1,1),reverseattr(ATTR,REV), field_attr(1,1,MAXLEN,REV),
ENDROW=NOOFROWS-1,
ENDCOL=NOOFCOLS-1,
line_hor(1,ENDCOL,0),
line_hor(1,ENDCOL,ENDROW),
line_ver(1,ENDROW,0),
line_ver(1,ENDROW,ENDCOL),
scr_char(ENDROW,0,'.'),
scr_char(ENDROW,ENDCOL,'.'),
lcorn(COL,LCORN), scr_char(0,0,LCORN),
RCOL=ACOL+ENDCOL,
rcorn(RCOL,RCORN), scr_char(0,ENDCOL,RCORN).

/* draw pulldown window corners */
  lcorn(0,'I') :- !.
  lcorn(_,'-').

  rcorn(79,'I') :- !.
  rcorn(_,'-').

  check_removewindow(0):-!.
  check_removewindow(_):-removewindow.

  is_up(up,_):-!.
  is_up(_,0).

  intense(ATTR,ATTR1):-
bitxor(ATTR,$08,ATTR1).

  intensefirstupper(ROW,COL,ATTR,WORD):-
frontchar(WORD,CH,_),
CH>='A', CH<='Z',!,scr_attr(ROW,COL,ATTR).
  intensefirstupper(ROW,COL,ATTR,WORD):-
frontchar(WORD,_,REST),COL1=COL+1,
intensefirstupper(ROW,COL1,ATTR,REST).

  intenseletter(ROW,COL,ATTR,WORD):-
intense(ATTR,INTENS),
intensefirstupper(ROW,COL,INTENS,WORD),!.
  intenseletter(ROW,COL,ATTR,_):-
intense(ATTR,INTENS),
scr_attr(ROW,COL,INTENS).

  pdwlist_strlist([],[]).
  pdwlist_strlist([curtain(_,H,_)|RESTPDW],[H|RESTSTR]):-
pdwlist_strlist(RESTPDW,RESTSTR).

  pdwmovevert(COL1,COL2,ATTR,LIST):-
pdwindex(COL1,LIST,curtain(POS1,WORD1,_)),str_len(WORD1,LEN1),
pdwindex(COL2,LIST,curtain(POS2,WORD2,_)),str_len(WORD2,LEN2),
field_attr(0,POS1,LEN1,ATTR),
intenseletter(0,POS1,ATTR,WORD1),
reverseattr(ATTR,REV),
field_attr(0,POS2,LEN2,REV),
intenseletter(0,POS2,REV,WORD2),
cursor(0,POS2).

  pdwlistlen([],0).
  pdwlistlen([_|T],N):-
pdwlistlen(T,X),
N=X+1.

  writepdwlist(_,[]).
  writepdwlist(ATTR,[curtain(POS,WORD,_)|T]):-
str_len(WORD,LEN),
field_str(0,POS,LEN,WORD),
intenseletter(0,POS,ATTR,WORD),
writepdwlist(ATTR,T).

  writelistp(_,_,_,[]).
  writelistp(ROW,LEN,ATTR,[H|T]):-
field_str(ROW,1,LEN,H),
intenseletter(ROW,1,ATTR,H),
ROW1=ROW+1,
writelistp(ROW1,LEN,ATTR,T).

  pdwindex(0,[H|_],H):-!.
  pdwindex(N,[_|T],X):-N1=N-1,pdwindex(N1,T,X).

  changepdwstate(_):-retract(pdwstate(_,_,_,_,_)),fail.
  changepdwstate(T):-assert(T).

  nextcol(0,-1,COL1,MAX):-COL1=MAX-1,!.
  nextcol(COL,1,0,MAX):-COL=MAX-1,!.
  nextcol(COL,DD,COL1,_):-COL1=COL+DD.

  pulldown(ATTR,LIST,CH1,CH2):-
textmode(_,C),
makewindow(81,ATTR,ATTR,"",0,0,3,C),
pdwlistlen(LIST,MAXCOL),
writepdwlist(ATTR,LIST),
pdwmovevert(0,0,ATTR,LIST),
changepdwstate(pdwstate(0,0,up,0,0)),
repeat,
pdwstate(ROW,COL,DOWN,MAXROW,LEN),
readkey(KEY),
pdwkeyact(KEY,ROW,COL,DOWN,MAXROW,MAXCOL,LEN,ATTR,LIST,CONTINUE),
CONTINUE=stop,removewindow,
pdwstate(ROW1,COL1,_,_,_),!,
CH1=COL1+1,
CH2=ROW1.

/*  Pulldown window action corresponding to input key and Pulldown window
    state */
  pdwkeyact(right,ROW,COL,up,MAXROW,MAXCOL,LEN,ATTR,LIST,cont):-
nextcol(COL,1,COL1,MAXCOL),
pdwmovevert(COL,COL1,ATTR,LIST),
changepdwstate(pdwstate(ROW,COL1,up,MAXROW,LEN)).

  pdwkeyact(right,ROW,COL,down,_,MAXCOL,_,ATTR,LIST,cont):-
nextcol(COL,1,COL1,MAXCOL),
check_removewindow(ROW),
pdwmovevert(COL,COL1,ATTR,LIST),
makepdwwindow(COL1,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
changepdwstate(pdwstate(FIRSTROW,COL1,down,MAXROW1,LEN1)).

  pdwkeyact(left,ROW,COL,up,MAXROW,MAXCOL,LEN,ATTR,LIST,cont):-
nextcol(COL,-1,COL1,MAXCOL),
pdwmovevert(COL,COL1,ATTR,LIST),
changepdwstate(pdwstate(ROW,COL1,up,MAXROW,LEN)).

  pdwkeyact(left,ROW,COL,down,_,MAXCOL,_,ATTR,LIST,cont):-
nextcol(COL,-1,COL1,MAXCOL),
check_removewindow(ROW),
pdwmovevert(COL,COL1,ATTR,LIST),
makepdwwindow(COL1,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
changepdwstate(pdwstate(FIRSTROW,COL1,down,MAXROW1,LEN1)).

  pdwkeyact(up,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,cont):-
ROW>1,!,
ROW1=ROW-1,
field_attr(ROW,1,LEN,ATTR),
pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
pdwindex(ROW1,LIST,WORD),
intenseletter(ROW,1,ATTR,WORD),
reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
cursor(ROW1,1),
changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).

  pdwkeyact(down,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,cont):-
ROW<MAXROW,!,
ROW1=ROW+1,
field_attr(ROW,1,LEN,ATTR),
pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
INDX=ROW-1,pdwindex(INDX,LIST,WORD),
intenseletter(ROW,1,ATTR,WORD),
reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
cursor(ROW1,1),
changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).

  pdwkeyact(down,_,COL,up,_,_,_,ATTR,LIST,cont):-
makepdwwindow(COL,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
changepdwstate(pdwstate(FIRSTROW,COL,down,MAXROW1,LEN1)).

  pdwkeyact(cr,_,COL,up,_,_,_,ATTR,LIST,stop):-
```

```
makepdwwindow(COL,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
changepdwstate(pdwstate(FIRSTROW,COL,down,MAXROW1,LEN1)),
FIRSTROW=0,
CH=COL+1, SUBCH=0,
not(pdwaction(CH,SUBCH)).

  pdwkeyact(cr,ROW,COL,down,_,_,_,_,_,stop):-
CH=COL+1, SUBCH=ROW,
not(pdwaction(CH,SUBCH)),
check_removewindow(ROW).

  pdwkeyact(char(CHAR),ROW,COL,UP,_,_,_,ATTR,PDWLIST,stop):-
is_up(UP,ROW),!,
pdwlist_strlist(PDWLIST,STRLIST),
tryletter(CHAR,STRLIST,SEL),NEWCOL=SEL,
pdwmovevert(COL,NEWCOL,ATTR,PDWLIST),
makepdwwindow(NEWCOL,ATTR,PDWLIST,MAXROW1,LEN1,FIRSTROW),
changepdwstate(pdwstate(FIRSTROW,NEWCOL,down,MAXROW1,LEN1)),
FIRSTROW=0,
CH=NEWCOL+1, SUBCH=0,
not(pdwaction(CH,SUBCH)).

  pdwkeyact(char(CHAR),ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,stop):-
ROW><0,
pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
tryletter(CHAR,LIST,SEL),ROW1=SEL+1,
field_attr(ROW,1,LEN,ATTR),
R=ROW-1,
pdwindex(R,LIST,OLDWORD),
intenseletter(ROW,1,ATTR,OLDWORD),
reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
cursor(ROW1,1),
CH=COL+1, SUBCH=ROW1,
changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)),
not(pdwaction(CH,SUBCH)),
removewindow.

  pdwkeyact(esc,ROW,COL,down,_,_,_,_,_,cont):-
check_removewindow(ROW),
changepdwstate(pdwstate(0,COL,up,0,0)).

/*pdwkeyact(fkey(1),_,_,_,_,_,_,_,_,cont):-help.
If a help system is used*/
```

G.2.6 INCLUDE lineinp.pro

```
/**********************************************************************/
/*                                                                    */
/*    Turbo Prolog Toolbox                                            */
/*    (C) Copyright 1987 Borland International.                       */
/*                                                                    */
```

```
 /* LINE INPUT DRIVER     */
 /*********************************************************************/

/* These two database predicates must be declared somewhere
in the program.

DATABASE
  insmode
  lineinpstate(STRING,COL)
  lineinpflag
*/

PREDICATES
  lineinput(ROW,COL,LEN,ATTR,ATTR,STRING,STRING,STRING)
  lineinput_leave(ROW,COL,LEN,ATTR,ATTR,STRING,STRING,STRING)
  nondeterm lineinput_repeat(ROW,COL,LEN,ATTR,ATTR,STRING,STRING,STRING)
  nondeterm lineinput_repeat1(ROW,COL,LEN,ATTR,ATTR,STRING,STRING,STRING
  ,STRING)
  lineinput1(ROW,COL,LEN,ATTR,ATTR,STRING,STRING,STRING,KEY)
  inpkeyact(KEY,COL,LEN,COL,COL,STRING,STRING)
  inpendkey(KEY)
  changemode
  inpstate(DBASEDOM)
  setlineinpflag(KEY)
  disp_str(COL,COL,LEN,STRING)
  lineinpcursor(COL,LEN,COL,COL,STRING)
  lin(KEY,COL,STRING,STRING)
  myfrontstr(COL,STRING,STRING,STRING)
  dropchar(STRING,STRING)

CLAUSES
/* Lineinput creates an input line at ROW and COL of LENGTH with a prompt
   CONSTTXT and an starting value OLDTXT that will be returned as NEWTXT
   if no new text is entered   */
  lineinput(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT):-
lineinput1(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT,KEY),
removewindow,     /* Removes window */
not(KEY=esc).

  lineinput_leave(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT):-
lineinput1(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT,KEY),
not(KEY=esc),!.
  lineinput_leave(_,_,_,_,_,_,_,_):-removewindow,fail.

/* A fail will cause backtracking to lineinp_repeat and a new lineinput
   will be displayed */
  lineinput_repeat(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT):-
lineinput1(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT1,KEY),
not(KEY=esc),!,
lineinput_repeat1(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT1,NEWTXT).
  lineinput_repeat(_,_,_,_,_,_,_,_):-removewindow,fail.
  lineinput_repeat1(_,_,_,_,_,_,_,TXT,TXT).
  lineinput_repeat1(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,_,NEWTXT):-
removewindow,
lineinput_repeat(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT).


  lineinput1(_,_,_,_,_,_,_,_,_):-retract(lineinpflag),fail.
  lineinput1(ROW,COL,LEN,WATTR,FATTR,CONSTTXT,OLDTXT,NEWTXT,KEY):-
MAXCOL=LEN,
adjframe(FATTR,1,MAXCOL,DR,DC),
adjustwindow(ROW,COL,DR,DC,ROW1,COL1),
makewindow(81,WATTR,FATTR,"",ROW1,COL1,DR,DC),
str_len(CONSTTXT,MINCOL),
write(CONSTTXT),
inpstate(lineinpstate(OLDTXT,0)),
REST=LEN-MINCOL,
repeat,
lineinpstate(CURTEXT,OFFSET),
disp_str(OFFSET,MINCOL,REST,CURTEXT),
cursor(_,CC),
readkey(KEY),          /* In TPREDS.PRO returns the tokenized input */
setlineinpflag(KEY),
inpkeyact(KEY,CC,MAXCOL,MINCOL,OFFSET,CURTEXT,OLDTXT),
inpendkey(KEY),
lineinpstate(NEWTXT,_),!.

  inpendkey(fkey(10)):-!.
  inpendkey(cr):-!.
  inpendkey(esc).

/* assert lineinpflag when control character is entered */
  setlineinpflag(char(_)):-!.
  setlineinpflag(_):-lineinpflag,!.
  setlineinpflag(_):-assert(lineinpflag).

  disp_str(_,_,_,_):-keypressed,!.
  disp_str(0,MINCOL,LEN,TEXT):-!,
field_str(0,MINCOL,LEN,TEXT).
  disp_str(OFFSET,MINCOL,LEN,TEXT):-
frontstr(OFFSET,TEXT,_,TXT2),!,
field_str(0,MINCOL,LEN,TXT2).
  disp_str(_,MINCOL,LEN,_):-
field_str(0,MINCOL,LEN,"").

  inpstate(_):-retract(lineinpstate(_,_)),fail.
  inpstate(T):-assert(T).

  lineinpcursor(COL,MAXCOL,MINCOL,_,_):-
COL<MAXCOL,COL>=MINCOL,!,cursor(0,COL).
  lineinpcursor(COL,MAXCOL,_,OFFSET,TXT):-
```

```
COL>=MAXCOL,!,
OFFSET1=1+OFFSET+COL-MAXCOL,
inpstate(lineinpstate(TXT,OFFSET1)),
COL1=MAXCOL-1,
cursor(0,COL1).
  lineinpcursor(COL,_,MINCOL,OFFSET,TXT):-
COL<MINCOL,
OFFSET1=OFFSET-(1+MINCOL-COL),
max(0,OFFSET1,OFFSET2),
inpstate(lineinpstate(TXT,OFFSET2)),
cursor(0,MINCOL).

  myfrontstr(N,STR,S1,S2):-frontstr(N,STR,S1,S2),!.
  myfrontstr(N,STR,S1,""):-
str_len(STR,NN),
LEN=N-NN,
str_len(SS,LEN),
concat(STR,SS,S1).

  changemode:-retract(insmode),!.
  changemode:-assert(insmode).

  lin(char(T),POS,STR,STR1):-
insmode,!,
myfrontstr(POS,STR,S1,S2),
frontchar(S22,T,S2),
concat(S1,S22,STR1).

  lin(char(T),POS,STR,STR1):-
myfrontstr(POS,STR,S1,S2),
dropchar(S2,S21),
frontchar(S22,T,S21),
concat(S1,S22,STR1).

  lin(del,POS,STR,STR1):-
frontstr(POS,STR,S1,S2),
frontchar(S2,_,S22),!,
concat(S1,S22,STR1).
  lin(del,_,S,S).

  dropchar(S,S1):-frontchar(S,_,S1),!.
  dropchar(S,S).


 /*********************************************************************
 inpkeyact carries out associated action for user input. Handles all
 standard input keys: arrow keys, insert, delete, etc.
 *********************************************************************/

  inpkeyact(char(T),COL,MAXCOL,MINCOL,OFFSET,TXT,OLDTXT):-
lineinpflag,!,
POS=OFFSET+COL-MINCOL,
lin(char(T),POS,TXT,TXT1),
inpstate(lineinpstate(TXT1,OFFSET)),
inpkeyact(right,COL,MAXCOL,MINCOL,OFFSET,TXT1,OLDTXT).

  inpkeyact(char(T),_,MAXCOL,MINCOL,_,_,OLDTXT):-
assert(lineinpflag),
str_char(TXT1,T),
inpstate(lineinpstate(TXT1,0)),
inpkeyact(right,MINCOL,MAXCOL,MINCOL,0,TXT1,OLDTXT).

  inpkeyact(del,COL,_,MINCOL,OFFSET,TXT,_):-
POS=OFFSET+COL-MINCOL,
lin(del,POS,TXT,TXT1),
inpstate(lineinpstate(TXT1,OFFSET)).

  inpkeyact(bdel,COL,MAXCOL,MINCOL,OFFSET,TXT,OLDTXT):-
COL>MINCOL,
POS=OFFSET+COL-1-MINCOL,
lin(del,POS,TXT,TXT1),
inpstate(lineinpstate(TXT1,OFFSET)),
inpkeyact(left,COL,MAXCOL,MINCOL,OFFSET,TXT1,OLDTXT).

  inpkeyact(cr,_,_,_,_,_,_).

  inpkeyact(fkey(10),_,_,_,_,_,_).

/*inpkeyact(fkey(1),_,_,_,_,_,_):-help.  If a help system is used  */

  inpkeyact(esc,_,_,_,_,_,_).

  inpkeyact(ins,_,_,_,_,_,_):-changemode.

  inpkeyact(home,_,_,MINCOL,_,TXT,_):-
inpstate(lineinpstate(TXT,0)),
cursor(0,MINCOL).

  inpkeyact(end,_,MAXCOL,MINCOL,_,TXT,_):-
str_len(TXT,STRLEN),
COL=MINCOL+STRLEN,
COL<MAXCOL,!,
inpstate(lineinpstate(TXT,0)),
cursor(0,COL).

  inpkeyact(end,_,MAXCOL,MINCOL,_,TXT,_):-
str_len(TXT,STRLEN),
OFFSET=(MINCOL+STRLEN+1)-MAXCOL,
inpstate(lineinpstate(TXT,OFFSET)),
COL=MAXCOL-1,
cursor(0,COL).
```

```
  inpkeyact(right,COL,MAXCOL,MINCOL,OFFSET,TXT,_):-
COL1=COL+1,
lineinpcursor(COL1,MAXCOL,MINCOL,OFFSET,TXT).

  inpkeyact(ctrlright,COL,MAXCOL,MINCOL,OFFSET,TXT,_):-
COL1=COL+5,
lineinpcursor(COL1,MAXCOL,MINCOL,OFFSET,TXT).

  inpkeyact(left,COL,MAXCOL,MINCOL,OFFSET,TXT,_):-
COL1=COL-1,
lineinpcursor(COL1,MAXCOL,MINCOL,OFFSET,TXT).

  inpkeyact(ctrlleft,COL,MAXCOL,MINCOL,OFFSET,TXT,_):-
COL1=COL-5,
lineinpcursor(COL1,MAXCOL,MINCOL,OFFSET,TXT).

  inpkeyact(ctrlbdel,_,_,MINCOL,_,_,_):-
inpstate(lineinpstate("",0)),
cursor(0,MINCOL).

  inpkeyact(fkey(8),_,_,MINCOL,_,_,OLDTXT):-
OLDTXT><"",
inpstate(lineinpstate(OLDTXT,0)),
cursor(0,MINCOL).
```

G.2.7 INCLUDE filename.pro

```
 /******************************************************************

    Turbo Prolog Toolbox
    (C) Copyright 1987 Borland International.

Read filename

    readfilename(ROW,COL,ATTR,EXTENSION,OLDFILENAME,NEWFILENAME)
 ******************************************************************/

/* These declarations and inclusions are necessary to try filename

DOMAINS
include "tdoms.pro"

DATABASE
  insmode
  lineinpstate(STRING,COL)
  lineinpflag

include "tpreds.pro"
include "lineinp.pro"
*/
```

```
PREDICATES
  readfilename(ROW,COL,ATTR,ATTR,STRING,STRING,STRING)
  readfilename1(ROW,COL,ATTR,ATTR,STRING,STRING,STRING)
  readfilename2(ROW,COL,ATTR,ATTR,STRING,STRING)
  newext(STRING,STRING,STRING)
  extentfilename(STRING,STRING,STRING)
  splitfilename(STRING,STRING,STRING)
  search_char(CHAR,STRING,INTEGER,INTEGER)
  concatlist(STRINGLIST,STRING)

CLAUSES
  readfilename(ROW,COL,WATTR,FATTR,EXT,OLD,FILENAME):-
concatlist(["File name (.",EXT,"): "],TXT),
lineinput_repeat(ROW,COL,40,WATTR,FATTR,TXT,OLD,NAME),
readfilename1(ROW,COL,WATTR,FATTR,EXT,NAME,FNAME),!,
removewindow,
extentfilename(FNAME,EXT,FILENAME).

  readfilename1(ROW,COL,WATTR,FATTR,EXT,"",FILENAME):-!,
ROW2=ROW+3,
readfilename2(ROW2,COL,WATTR,FATTR,EXT,FILENAME).
  readfilename1(_,_,_,_,_,NAME,NAME).

  readfilename2(ROW,COL,WATTR,FATTR,EXT,FILENAME):-
adjustwindow(ROW,COL,10,50,ROW1,COL1),
makewindow(81,WATTR,FATTR,"",ROW1,COL1,10,50),
concat("*.",EXT,EXT1),
dir("",EXT1,FILENAME),!,
removewindow.
  readfilename2(_,_,_,_,_,_):-removewindow,fail.

/* replace old extension with the new extension */
  newext(OLD,EXT,NEW):-
splitfilename(OLD,NAME,_),
concatlist([NAME,".",EXT],NEW).

  extentfilename(OLDNAME,EXT,FILENAME):-
splitfilename(OLDNAME,NAME,OLDEXT), OLDEXT="",!,
concatlist([NAME,".",EXT],FILENAME).
  extentfilename(NAME,_,NAME).

/* parse file name string for name and extension */
  splitfilename(FILENAME,NAME,EXT):-
search_char('.',FILENAME,0,N),
frontstr(N,FILENAME,NAME,REST),
frontchar(REST,_,EXT),!.
  splitfilename(NAME,NAME,"").

  search_char(CH,STR,N,N) :-frontchar(STR,CH,_),!.
  search_char(CH,STR,N,N2) :-
         frontchar(STR,_,STR1),
         N1=N+1,
         search_char(CH,STR1,N1,N2).

  concatlist([],"").
  concatlist([H|T],S):-
concatlist(T,S1),
concat(H,S1,S).


 /*********************************************************************/
 /* SET/CHANGE DIRECTORY     */
 /*********************************************************************/

PREDICATES
  setdir(ROW,COL,ATTR,ATTR)
  /* Make a window and prompt the user with the current
     directory and set the new directory to the new value
     if possable. */
  newdisk(STRING)
  /* change directory to DISK if it exists or return an error
     message */
CLAUSES
  setdir(ROW,COL,WATTR,FATTR):-
disk(DISK),
lineinput_repeat(ROW,COL,45,WATTR,FATTR,"Directory: ",DISK,NEWDISK),
newdisk(NEWDISK),!,
removewindow.
  setdir(_,_,_,_).

  newdisk(DISK):-disk(DISK),!.
  newdisk(_):- makewindow(1,7,7,"",10,30,4,45),
write(">> Error in directory name"),nl,
write("   - Press return"),
readkey(_),
removewindow,
fail.
```

G.2.8 INCLUDE datalog.sca

```
/************************************************************
            Turbo Prolog Toolbox
            (C) Copyright 1987 Borland International.

Some parts have their origin in the Toolbox
and others are my own production. Finn Pedersen. August 1991.
***********************************************************/

DOMAINS
/*CURSOR = integer  */
  CURSORTOK = t(TOK,CURSOR)
  TOKL = CURSORTOK*
```

```
/*
  PROCID = string     */

/* include "XDATALOG.DOM"  */

PREDICATES

/* scan_error(STRING,CURSOR)  */
  tokl(CURSOR,STRING,TOKL)
  new_fronttoken(string,string,string)
  maketok(CURSOR,STRING,TOK,STRING,STRING,CURSOR)
  str_tok(STRING,TOK)
  scan_str(CURSOR,STRING,STRING,STRING)
  search_ch(CHAR,STRING,INTEGER,INTEGER)
  skipspaces(STRING,STRING,INTEGER)
  skipcomment(STRING,STRING,INTEGER)
  isspace(CHAR)

CLAUSES

/* scan_error(_,_) :- write("No error recovery").  */
  tokl(POS,STR,[t(TOK,POS1)|TOKL]):-
skipspaces(STR,STR1,NOOFSP),
POS1=POS+NOOFSP,
new_fronttoken(STR1,STRTOK,STR2),!,
maketok(POS,STRTOK,TOK,STR2,STR3,LEN1),
str_len(STRTOK,LEN),
POS2=POS1+LEN+LEN1,
tokl(POS2,STR3,TOKL).
  tokl(_,_,[]).

  new_fronttoken(STR1,"<=",STR3)
     :- fronttoken(STR1,"<",STR2),
        fronttoken(STR2,"=",STR3),!.
  new_fronttoken(STR1,">=",STR3)
     :- fronttoken(STR1,">",STR2),
        fronttoken(STR2,"=",STR3),!.
  new_fronttoken(STR1,TOKEN,STR2)
     :- fronttoken(STR1,TOKEN,STR2).

  skipspaces(STR1,STR4,NOOFSP):-
frontchar(STR1,'%',STR2),!,
skipcomment(STR2,STR3,N1),!,
skipspaces(STR3,STR4,N2),
NOOFSP=N1+N2+1.

  skipspaces(STR,STR2,NOOFSP):-
frontchar(STR,CH,STR1),isspace(CH),!,
skipspaces(STR1,STR2,N1),
NOOFSP=N1+1.
  skipspaces(STR,STR,0).
  skipcomment(STR1,STR2,1) :-
frontchar(STR1,'%',STR2),!.
  skipcomment(STR1,STR3,N1) :-
frontchar(STR1,_,STR2),!,
skipcomment(STR2,STR3,N), N1 = N+1.

  isspace(' ').
  isspace('\t').
  isspace('\n').



  str_tok(",",comma):-!.
  str_tok("&",et):-!.
  str_tok("?",questionmark):-!.
  str_tok("-",minus):-!.
  str_tok(":",collon):-!.
  str_tok(";",semicollon):-!.
  str_tok(".",punctum):-!.
  str_tok("(",lpar):-!.
  str_tok(")",rpar):-!.
  str_tok("[",lshp):-!.
  str_tok("]",rshp):-!.
  str_tok("{",ltub):-!.
  str_tok("}",rtub):-!.
  str_tok("_",anonymousvariable):-!.
  str_tok("not",non):-!.
  str_tok("bb",pre_bound):-!.
  str_tok("fb",free_bound):-!.
  str_tok("ff",post_free):-!.
  str_tok("!",exclamationmark):-!.
  str_tok("=",equal):-!.
  str_tok("<",less):-!.
  str_tok(">",greater):-!.
  str_tok("<=",less_equal):-!.
  str_tok(">=",greater_equal):-!.
  str_tok("+",plus):-!.
  str_tok("*",mult):-!.
  str_tok("/",div):-!.


  maketok(_,STR,TOK,S,S,0):-str_tok(STR,TOK),!.
  maketok(CURSOR,"\"",str(STR),S1,S2,LEN):-!,
scan_str(CURSOR,S1,S2,STR),str_len(STR,LEN1),LEN=LEN1+1.
  maketok(_,INTSTR,int(INTEGER),S,S,0):-str_int(INTSTR,INTEGER),!.
  maketok(_,STR,var(STR),S,S,0):-frontchar(STR,CH,_),CH>='A',CH<='Z',!.
  maketok(_,STRING,id(STRING),S,S,0):-isname(STRING),!.
  maketok(CURSOR,_,_,_,_,_):-scan_error("Illegal token",CURSOR),fail.
  scan_str(_,IN,OUT,STR):-
search_ch('"',IN,0,N),
frontstr(N,IN,STR,OUT1),!,
frontchar(OUT1,_,OUT).
  scan_str(CURSOR,_,_,_):-scan_error("String not terminated",CURSOR),fail.


  search_ch(CH,STR,N,N):-
frontchar(STR,CH,_),!.
  search_ch(CH,STR,N,N1):-
frontchar(STR,_,S1),
N2=N+1,
search_ch(CH,S1,N2,N1).
```

G.2.9 INCLUDE datalog.par

```
/*********************************************************
PARSING PREDICATES
*********************************************************/

PREDICATES
  s_dbase(TOKL,TOKL,Dbase)
  s_adornlist(TOKL,TOKL,AdornList)
  s_adornlist1(TOKL,TOKL,AdornList)
  s_termlist(TOKL,TOKL,Termlist)
  s_termlist1(TOKL,TOKL,Termlist)
  s_body(TOKL,TOKL,Body)
  s_body1(TOKL,TOKL,Body)
  s_qtermlist(TOKL,TOKL,QTermlist)
  s_qtermlist1(TOKL,TOKL,QTermlist)
  s_datalog(TOKL,TOKL,Datalog)
  s_query(TOKL,TOKL,Query)
  s_rule(TOKL,TOKL,Rule)
  s_rule1(TOKL,TOKL,Head,Rule)
  s_rule2(TOKL,TOKL,STRING,Rule)
  s_adornbinding(TOKL,TOKL,AdornBinding)
  s_head(TOKL,TOKL,Head)
  s_head1(TOKL,TOKL,STRING,Head)
  s_literal(TOKL,TOKL,Literal)
  s_atom(TOKL,TOKL,Atom)
  s_atom1(TOKL,TOKL,STRING,Atom)
  s_atom2(TOKL,TOKL,Expr,Atom)
  s_expr(TOKL,TOKL,Expr)
  s_expr1(TOKL,TOKL,Expr)
  s_expr4(TOKL,TOKL,Expr,Expr)
  s_expr2(TOKL,TOKL,Expr)
  s_expr5(TOKL,TOKL,Expr,Expr)
  s_expr3(TOKL,TOKL,Expr)
  s_qliteral(TOKL,TOKL,QLiteral)
  s_qatom(TOKL,TOKL,QAtom)
  s_qatom1(TOKL,TOKL,STRING,QAtom)
  s_qexpr(TOKL,TOKL,QExpr)
  s_qexpr1(TOKL,TOKL,INTEGER,QExpr)
  s_number(TOKL,TOKL,Number)
  s_number1(TOKL,TOKL,INTEGER,Number)

CLAUSES
  s_datalog(LL1,LL0,datalog(Query,Dbase)):-
s_query(LL1,LL2,Query),
s_dbase(LL2,LL0,Dbase),!.

  s_query([t(questionmark,_)|LL1],LL0,query(QLiteral)):-!,
expect(t(minus,_),LL1,LL2),
s_qliteral(LL2,LL3,QLiteral),
expect(t(punctum,_),LL3,LL0).
  s_query(LL,_,_):-syntax_error(query,LL),fail.

  s_rule(LL1,LL0,Rule_):-
s_head(LL1,LL2,Head),
s_rule1(LL2,LL0,Head,Rule_),!.
  s_rule([t(id(STRING),_)|LL1],LL0,Rule_):-!,
s_rule2(LL1,LL0,STRING,Rule_).
  s_rule(LL,_,_):-syntax_error(rule,LL),fail.

  s_adornbinding([t(pre_bound,_)|LL],LL,bb):-!.
  s_adornbinding([t(free_bound,_)|LL],LL,fb):-!.
  s_adornbinding([t(post_free,_)|LL],LL,ff):-!.
  s_adornbinding(LL,_,_):-syntax_error(adornbinding,LL),fail.

  s_head([t(id(STRING),_)|LL1],LL0,Head_):-!,
s_head1(LL1,LL0,STRING,Head_).
  s_head(LL,_,_):-syntax_error(head,LL),fail.

  s_literal(LL1,LL0,pos(Atom)):-
s_atom(LL1,LL0,Atom),!.
  s_literal([t(non,_)|LL1],LL0,neg(Atom)):-!,
expect(t(lpar,_),LL1,LL2),
s_atom(LL2,LL3,Atom),
expect(t(rpar,_),LL3,LL0).
  s_literal(LL,_,_):-syntax_error(literal,LL),fail.

  s_atom([t(id(STRING),_)|LL1],LL0,Atom_):-
s_atom1(LL1,LL0,STRING,Atom_),!.
  s_atom(LL1,LL0,Atom_):-
s_expr(LL1,LL2,Expr),
s_atom2(LL2,LL0,Expr,Atom_),!.
  s_atom(LL,_,_):-syntax_error(atom,LL),fail.

  s_expr(LL1,LL0,Expr):-
s_expr1(LL1,LL0,Expr).
```

```
  s_expr1(LL1,LL0,Expr_):-
s_expr2(LL1,LL2,Expr),
s_expr4(LL2,LL0,Expr,Expr_).

  s_expr2(LL1,LL0,Expr_):-
s_expr3(LL1,LL2,Expr),
s_expr5(LL2,LL0,Expr,Expr_).

  s_expr3([t(var(STRING),_)|LL],LL,var(STRING)):-!.
  s_expr3([t(int(INTEGER),_)|LL],LL,int(INTEGER)):-!.
  s_expr3([t(str(STRING),_)|LL],LL,str(STRING)):-!.
  s_expr3([t(id(STRING),_)|LL],LL,str(STRING)):-!.
  s_expr3([t(anonymousvariable,_)|LL],LL,anonymousvariable):-!.
  s_expr3([t(minus,_)|LL1],LL0,sign(Expr)):-!,
s_expr3(LL1,LL0,Expr).
  s_expr3([t(lpar,_)|LL1],LL0,Expr):-!,
s_expr(LL1,LL2,Expr),
expect(t(rpar,_),LL2,LL0).
  s_expr3(LL,_,_):-syntax_error(expr3,LL),fail.

  s_qliteral(LL1,LL0,pos(QAtom)):-
s_qatom(LL1,LL0,QAtom),!.

  s_qatom([t(id(STRING),_)|LL1],LL0,QAtom_):-!,
s_qatom1(LL1,LL0,STRING,QAtom_).
  s_qatom(LL,_,_):-syntax_error(qatom,LL),fail.

  s_qexpr([t(var(STRING),_)|LL],LL,var(STRING)):-!.
  s_qexpr([t(minus,_)|LL1],LL0,signrat(INTEGER,INTEGER1)):-
expect(t(int(INTEGER),_),LL1,LL2),
expect(t(div,_),LL2,LL3),
expect(t(int(INTEGER1),_),LL3,LL0),!.
  s_qexpr([t(int(INTEGER),_)|LL1],LL0,QExpr_):-!,
s_qexpr1(LL1,LL0,INTEGER,QExpr_).
  s_qexpr([t(minus,_)|LL1],LL0,sign(INTEGER)):-!,
expect(t(int(INTEGER),_),LL1,LL0).
  s_qexpr([t(str(STRING),_)|LL],LL,str(STRING)):-!.
  s_qexpr([t(anonymousvariable,_)|LL],LL,anonymousvariable):-!.
  s_qexpr([t(id(STRING),_)|LL],LL,str(STRING)):-!.
  s_qexpr(LL,_,_):-syntax_error(qexpr,LL),fail.

  s_number([t(minus,_)|LL1],LL0,signrat(INTEGER,INTEGER1)):-
expect(t(int(INTEGER),_),LL1,LL2),
expect(t(div,_),LL2,LL3),
expect(t(int(INTEGER1),_),LL3,LL0),!.
  s_number([t(int(INTEGER),_)|LL1],LL0,Number_):-!,
s_number1(LL1,LL0,INTEGER,Number_).
  s_number([t(minus,_)|LL1],LL0,sign(INTEGER)):-!,
expect(t(int(INTEGER),_),LL1,LL0).
  s_number(LL,_,_):-syntax_error(number,LL),fail.
  s_rule1([t(collon,_)|LL1],LL0,Head,rule(Head,Body)):-!,
expect(t(minus,_),LL1,LL2),
s_body(LL2,LL3,Body),
expect(t(punctum,_),LL3,LL0).
  s_rule1([t(punctum,_)|LL],LL,Head,rule2(Head)):-!.
  s_rule1(LL,_,_,_):-syntax_error(rule1,LL),fail.

  s_rule2([t(lshp,_)|LL1],LL0,STRING,
  range(STRING,INTEGER,Number,Number1)):-!,
expect(t(int(INTEGER),_),LL1,LL2),
expect(t(rshp,_),LL2,LL3),
expect(t(equal,_),LL3,LL4),
expect(t(lshp,_),LL4,LL5),
s_number(LL5,LL6,Number),
expect(t(semicollon,_),LL6,LL7),
s_number(LL7,LL8,Number1),
expect(t(rshp,_),LL8,LL9),
expect(t(punctum,_),LL9,LL0).
  s_rule2([t(collon,_)|LL1],LL0,STRING,safety(STRING,INTEGER)):-!,
expect(t(int(INTEGER),_),LL1,LL2),
expect(t(punctum,_),LL2,LL0).
  s_rule2([t(ltub,_)|LL1],LL0,STRING,adornment(STRING,AdornList)):-!,
s_adornlist(LL1,LL2,AdornList),
expect(t(rtub,_),LL2,LL3),
expect(t(punctum,_),LL3,LL0).
  s_rule2(LL,_,_,_):-syntax_error(rule2,LL),fail.

  s_head1([t(lpar,_)|LL1],LL0,STRING,head(STRING,Termlist)):-!,
s_termlist(LL1,LL2,Termlist),
expect(t(rpar,_),LL2,LL0).
  s_head1(LL,LL,STRING,head2(STRING)):-!.

  s_atom1([t(lpar,_)|LL1],LL0,STRING,head(STRING,Termlist)):-!,
s_termlist(LL1,LL2,Termlist),
expect(t(rpar,_),LL2,LL0).
  s_atom1(LL,LL,STRING,head2(STRING)):-!.

  s_atom2([t(less,_)|LL1],LL0,Expr,less_than(Expr,Expr1)):-!,
s_expr(LL1,LL0,Expr1).
  s_atom2([t(less_equal,_)|LL1],LL0,Expr,less_equal(Expr,Expr1)):-!,
s_expr(LL1,LL0,Expr1).
  s_atom2([t(greater_equal,_)|LL1],LL0,Expr,greater_equal(Expr,Expr1)):-!,
s_expr(LL1,LL0,Expr1).
  s_atom2([t(greater,_)|LL1],LL0,Expr,greater_than(Expr,Expr1)):-!,
s_expr(LL1,LL0,Expr1).
  s_atom2([t(exclamationmark,_)|LL1],LL0,Expr,not_equal(Expr,Expr1)):-!,
expect(t(equal,_),LL1,LL2),
s_expr(LL2,LL0,Expr1).
  s_atom2([t(equal,_)|LL1],LL0,Expr,equal(Expr,Expr1)):-!,
s_expr(LL1,LL0,Expr1).
  s_atom2(LL,_,_,_):-syntax_error(atom2,LL),fail.
  s_expr4([t(plus,_)|LL1],LL0,Expr,Expr_):-!,
s_expr2(LL1,LL2,Expr1),
s_expr4(LL2,LL0,plus(Expr,Expr1),Expr_).
  s_expr4([t(minus,_)|LL1],LL0,Expr,Expr_):-!,
s_expr2(LL1,LL2,Expr1),
s_expr4(LL2,LL0,minus(Expr,Expr1),Expr_).
  s_expr4(LL,LL,Expr,Expr).

  s_expr5([t(mult,_)|LL1],LL0,Expr,Expr_):-!,
s_expr3(LL1,LL2,Expr1),
s_expr5(LL2,LL0,mult(Expr,Expr1),Expr_).
  s_expr5([t(div,_)|LL1],LL0,Expr,Expr_):-!,
s_expr3(LL1,LL2,Expr1),
s_expr5(LL2,LL0,div(Expr,Expr1),Expr_).
  s_expr5(LL,LL,Expr,Expr).

  s_qatom1([t(lpar,_)|LL1],LL0,STRING,head(STRING,QTermlist)):-!,
s_qtermlist(LL1,LL2,QTermlist),
expect(t(rpar,_),LL2,LL0).
  s_qatom1(LL,LL,STRING,head2(STRING)):-!.

  s_qexpr1([t(div,_)|LL1],LL0,INTEGER,rat(INTEGER,INTEGER1)):-!,
expect(t(int(INTEGER1),_),LL1,LL0).
  s_qexpr1(LL,LL,INTEGER,int(INTEGER)):-!.

  s_number1([t(div,_)|LL1],LL0,INTEGER,rat(INTEGER,INTEGER1)):-!,
expect(t(int(INTEGER1),_),LL1,LL0).
  s_number1(LL,LL,INTEGER,int(INTEGER)):-!.

  s_dbase(LL1,LL0,[Rule|Dbase]):-
s_rule(LL1,LL2,Rule),!,
s_dbase(LL2,LL0,Dbase).
  s_dbase(LL,LL,[]).

  s_adornlist(LL1,LL0,[AdornBinding|AdornList]):-
s_adornbinding(LL1,LL2,AdornBinding),
s_adornlist1(LL2,LL0,AdornList).

  s_adornlist1([t(comma,_)|LL1],LL2,AdornList):-!,
s_adornlist(LL1,LL2,AdornList).
  s_adornlist1(LL,LL,[]).

  s_termlist(LL1,LL0,[Expr|Termlist]):-
s_expr(LL1,LL2,Expr),
s_termlist1(LL2,LL0,Termlist).

  s_termlist1([t(comma,_)|LL1],LL2,Termlist):-!,
s_termlist(LL1,LL2,Termlist).
  s_termlist1(LL,LL,[]).
  s_body(LL1,LL0,[Literal|Body]):-
s_literal(LL1,LL2,Literal),
s_body1(LL2,LL0,Body).

  s_body1([t(et,_)|LL1],LL2,Body):-!,
s_body(LL1,LL2,Body).
  s_body1(LL,LL,[]).

  s_qtermlist(LL1,LL0,[QExpr|QTermlist]):-
s_qexpr(LL1,LL2,QExpr),
s_qtermlist1(LL2,LL0,QTermlist).

  s_qtermlist1([t(comma,_)|LL1],LL2,QTermlist):-!,
s_qtermlist(LL1,LL2,QTermlist).
  s_qtermlist1(LL,LL,[]).
```

G.3 MODULE mat.pro

```
ifndef localcompilation
project "datalog"
include "globals.pro"
enddef
domains
  INTS  = INTEGER*
  Table = F*
  F = f(INTEGER,INTEGER)

constants
primelist = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
             53,59,61,67,71,73,79,83,89,97]


predicates

  gcd (INTEGER,INTEGER,INTEGER)

  union_factor (Table,Table,Table)
  insert_factor (F,Table,Table)
  max (INTEGER,INTEGER,INTEGER)

  reduce_factors (Table,Table,Table)
  reduce_fraction (INTEGER,INTEGER,INTEGER,INTEGER)


  unfactorize (Table,INTEGER)
  factorial (INTEGER,INTEGER,INTEGER)


  factorize (INTEGER,Table)
  factorize2 (Table,INTEGER,Table)
  smallestfactor (INTS,INTEGER,INTEGER)
  insertfactor (INTEGER,Table,Table)
/*
  primenumbers (INTS)
*/

%  times (Expr,Expr,Expr)
%  divide (Expr,Expr,Expr)
%  plus (Expr,Expr,Expr)
  sign (INTEGER,INTEGER)
%  minus (Expr,Expr,Expr)


%  less_than (Expr,Expr)
%  less_equal (Expr,Expr)
%  not_equal (Expr,Expr)
%  equal (Expr,Expr)

%relational numbers to floating point numbers

%  rational2real (Expr,Real)
%  number2real (Number,Real)


clauses

%greatest common devisor
  gcd(0,_,1) :- !.
  gcd(I,I,I) :- !.
  gcd(I,J,K) :- I > J, !, IJ = I-J, gcd(IJ,J,K), !.
  gcd(I,J,K) :- JI = J-I, gcd(I,JI,K).



  union_factor ([],B,B) :- !.
  union_factor (B,[],B) :- !.
  union_factor (A,[T|S1],C) :- union_factor(A,S1,S2),
      insert_factor(T,S2,C).


  insert_factor (T,[],[T]) :- !.
  insert_factor (f(P,N),[f(P,M)|R],[f(P,Max)|R]) :- max(N,M,Max), !.
  insert_factor (f(P,N),[f(Q,M)|R1],[f(Q,M)|R2])
     :- P > Q, insert_factor(f(P,N),R1,R2),!.
  insert_factor (f(P,N),R,[f(P,N)|R]).




  max(A,B,A) :- A >= B, !.
  max(_,B,B).



  reduce_factors(RF,[],RF) :- !.
  reduce_factors([],RF,RF) :- !.
  reduce_factors([f(P,N1)|F1],[f(P,N2)|F2],[f(P,N4)|F3])
     :- N3 = N1 - N2, N4 = abs(N3), not(N4=0),
        reduce_factors(F1,F2,F3),!.
  reduce_factors([f(P,N)|F1],[f(P,N)|F2],F3)
     :- reduce_factors(F1,F2,F3),!.
  reduce_factors([f(P,N1)|F1],[f(Q,N2)|F2],[f(Q,N2)|F3])
     :- P > Q,
        reduce_factors([f(P,N1)|F1],F2,F3),!.
  reduce_factors([f(P,N1)|F1],[f(Q,N2)|F2],[f(P,N1)|F3])
     :- P < Q,
        reduce_factors(F1,[f(Q,N2)|F2],F3),!.



  reduce_fraction(0,_,0,1) :- !.
  reduce_fraction(Numerator,Denominator,NumReduced,DenReduced)
     :- gcd(Numerator,Denominator,GCD),
        factorize(GCD,GCDF),
        factorize(Numerator,NF),
        factorize(Denominator,DF),
        reduce_factors(NF,GCDF,NFR),
        reduce_factors(DF,GCDF,DFR),
        unfactorize(NFR,NumReduced),
        unfactorize(DFR,DenReduced).




  unfactorize([],1) :- !.
  unfactorize([f(P,N)|F],K)
     :- unfactorize(F,I),
        factorial(P,N,J),
        K = I * J,!.


  factorial(P,1,P) :- !.
  factorial(P,N,R)
     :- N1 = N-1,
        factorial(P,N1,Q),
        R = Q * P.


  factorize(N,Table) :- factorize2([],N,Table).
```

```
  factorize2(Table,1,Table) :- !.
  factorize2(Table1,N,Table3)
     :- /*primenumbers(P),*/
        smallestfactor(primelist,N,M),
        insertfactor(M,Table1,Table2),
        N1 = N div M,
        factorize2(Table2,N1,Table3).


  smallestfactor([P|_],N,P)
     :- 0 = N mod P,!.
  smallestfactor([_|REST],N,P)
     :- smallestfactor(REST,N,P),!.
  smallestfactor([],N,N).


  insertfactor(M,[],[f(M,1)]) :- !.
  insertfactor(M,[f(M,I)|REST],[f(M,I1)|REST]) :- I1 = I + 1, !.
  insertfactor(M,[F|REST1],[F|REST2])
     :- insertfactor(M,REST1,REST2).

/*
primenumbers(L) :- L =
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,
79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,
163,167,173,179,181,191,193,197,199,211,223,227,229,233,239,
241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,
337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,
431,433,439,443,449,457,461,463,467,479,487,491,499,503,509,
521,523,541,547,557,563,569,571,577,587,593,599,601,607,613,
617,619,631,641,643,647,653,659,661,673,677,683,691,701,709,
719,727,733,739,743,751,757,761,769,773,787,797,809,811,821,
823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,
929,937,941,947,953,967,971,977,983,991,997].
*/

times(q(S1,T1,N1),q(S2,T2,N2),q(S3,T4,N4))
   :- S3 = S1 * S2,
      T3 = T1 * T2,
      N3 = N1 * N2,
      reduce_fraction(T3,N3,T4,N4).

divide(q(S1,T1,N1),q(S2,T2,N2),R)
   :- times(q(S1,T1,N1),q(S2,N2,T2),R).

plus(q(S1,T1,N1),q(S2,T2,N2),q(S3,T5,N5))
   :- factorize(N1,F1),
      factorize(N2,F2),
      union_factor(F1,F2,FN),
      reduce_factors(FN,F1,RF1),
      reduce_factors(FN,F2,RF2),
      unfactorize(RF1,R1),
      unfactorize(RF2,R2),
      unfactorize(FN,RN),

      T3 = S1*T1*R1 + S2*T2*R2,
      sign(T3,S3),
      T4 = abs(T3),

      reduce_fraction(T4,RN,T5,N5).


sign(T,1) :- T >= 0, !.
sign(_,-1).


minus(R1,q(S2,T2,N2),R3)
   :- MS = -S2, plus(R1,q(MS,T2,N2),R3).


less_than(_,anonymousvariable()) :- !.
less_than(str(A),str(B)) :- A < B,!.
less_than(q(_,_,_),str(_)) :- !.
less_than(R1,R2) :- minus(R1,R2,R3), R3 = q(-1,_,_),!.

less_equal(str(A),str(B)) :- A <= B,!.
less_equal(R1,R2) :- minus(R1,R2,R3), R3 = q(-1,_,_),!.
less_equal(R1,R2) :- minus(R1,R2,R3), R3 = q(_,0,_),!.

not_equal(str(A),str(B)) :- not(A = B),!.
not_equal(R1,R2) :- minus(R1,R2,R3), R3 = q(_,T,_), not(T=0),!.

equal(E,E) :- !.
equal(R1,R2) :- minus(R1,R2,R3), R3 = q(_,0,_),!.


rational2real(q(S,N,D),R):- R = S*N/D.

number2real(signrat(N,D),R) :- R = -N/D,!.
number2real(rat(N,D),R) :- R = N/D,!.
number2real(sign(N),R) :- R = -N,!.
number2real(int(N),R) :- R = N,!.
```

G.4 MODULE qconvert.pro

```
project "datalog"
include "globals.pro"

PREDICATES
```

```
%convert_rules (Dbase,Dbase)
convert_rule (Rule,Rule)
convert_body (Body,Body)
convert_literal (Literal,Literal)
convert_atom (Atom,Atom)
convert_termlist(Termlist,Termlist)
convert_expr (Expr,Expr).


CLAUSES
convert_rules([Rule1|Rules1],[Rule2|Rules2])
   :- convert_rule(Rule1,Rule2),
      convert_rules(Rules1,Rules2),!.
convert_rules([],[]).


convert_rule(rule(head(ID,TS1),BODY1),rule(head(ID,TS2),BODY2))
   :- convert_termlist(TS1,TS2),
      convert_body(BODY1,BODY2).

convert_body([Literal1|Body1],[Literal2|Body2])
   :- convert_literal(Literal1,Literal2),
      convert_body(Body1,Body2),!.
convert_body([],[]).

convert_literal(pos(Atom1),pos(Atom2))
   :- convert_atom(Atom1,Atom2),!.
convert_literal(neg(Atom1),neg(Atom2))
   :- convert_atom(Atom1,Atom2),!.

convert_atom(head(ID,TS1),head(ID,TS2))
   :- convert_termlist(TS1,TS2),!.
convert_atom(less_than(T1,T2),less_than(S1,S2))
   :- convert_expr(T1,S1),
      convert_expr(T2,S2),!.
convert_atom(less_equal(T1,T2),less_equal(S1,S2))
   :- convert_expr(T1,S1),
      convert_expr(T2,S2),!.
convert_atom(greater_than(T1,T2),greater_than(S1,S2))
   :- convert_expr(T1,S1),
      convert_expr(T2,S2),!.
convert_atom(greater_equal(T1,T2),greater_equal(S1,S2))
   :- convert_expr(T1,S1),
      convert_expr(T2,S2),!.
convert_atom(not_equal(T1,T2),not_equal(S1,S2))
   :- convert_expr(T1,S1),
      convert_expr(T2,S2),!.
convert_atom(equal(T1,T2),equal(S1,S2))
   :- convert_expr(T1,S1),
      convert_expr(T2,S2),!.
```

```
convert_termlist([QExpr|QTermlist],[Expr|Termlist])
   :- convert_expr(QExpr,Expr),
      convert_termlist(QTermlist,Termlist),!.
convert_termlist([],[]).

convert_expr(int(I),q(1,I,1)) :- !.
convert_expr(plus(E1,E2),plus(S1,S2))
   :- convert_expr(E1,S1),
      convert_expr(E2,S2),!.
convert_expr(minus(E1,E2),minus(S1,S2))
   :- convert_expr(E1,S1),
      convert_expr(E2,S2),!.
convert_expr(mult(E1,E2),mult(S1,S2))
   :- convert_expr(E1,S1),
      convert_expr(E2,S2),!.
convert_expr(div(E1,E2),div(S1,S2))
   :- convert_expr(E1,S1),
      convert_expr(E2,S2),!.
convert_expr(sign(E1),sign(S1))
   :- convert_expr(E1,S1),!.
convert_expr(E,E).
```

G.5 MODULE write.pro

```
ifndef localcompilation
project "datalog"
include "globals.pro"
enddef
PREDICATES
  writelist(Adornlist)

CLAUSES

vis_resultat2([]) :- nl,!.
vis_resultat2([H|R]) :- write(H), nl, vis_resultat2(R).

vis_resultat3([]) :- !.
vis_resultat3([init(H)|R])
:- write("INIT:",H), nl, vis_resultat3(R),!.
vis_resultat3([step(H)|R])
:- write("STEP:"), nl, vis_db(H), vis_resultat3(R),!.
vis_resultat3([repeat(H)|R])
:- write("REPEAT:"), nl, vis_db(H), vis_resultat3(R),!.

vis_all_facts
   :- fact_edb(ID,TS),
      write(ID), vis_termlist(TS), write("."), nl,
      fail.
```

```
vis_all_facts.

vis_all_range
   :- range_db(ID,Arg,Low,Upp),
      write(ID,"[",Arg,"]"," = ["),
      vis_number(Low),
      write(";"),
      vis_number(Upp),
      write("]."), nl,
      fail.
vis_all_range.


vis_all_safety
   :- safety_db (ID,Level),
      write(ID,":",Level,"."), nl,
      fail.
vis_all_safety.

vis_all_adornment
   :- adornment_db (ID,Adornlist),
      write(ID,"{"), writelist(Adornlist), write("}."), nl,
      fail.
vis_all_adornment.


vis_number(signrat(T,N)) :- write("-",T,"/",N),!.
vis_number(rat(T,N)) :- write(T,"/",N),!.
vis_number(sign(T)) :- write("-",T),!.
vis_number(int(T)) :- write(T).



writelist([])    :- !.
writelist([H])   :- write(H), !.
writelist([H|R]) :- write(H,","), writelist(R).





vis_db([]) :- !.
vis_db([Rule|DB])
   :- vis_rule(Rule),
      vis_db(DB).




vis_rule(rule(head(ID,TS),Body))
   :- write(ID), vis_termlist(TS), vis_body(Body).

vis_termlist([]) :- !.
vis_termlist([T|TS])
   :- write("("), vis_term(T), vis_termlist1(TS), write(")").

vis_termlist1([]) :- !.
vis_termlist1([T|TS])
   :- write(","), vis_term(T), vis_termlist1(TS).

vis_body([]) :- !, write("."), nl.
vis_body(Body)
   :- nl, write("   :-   "), vis_body1(Body), write("."), nl,nl.

vis_body1([L]) :- vis_literal(L), !.
vis_body1([L|Body])
   :- vis_literal(L),
      write(" &"), nl, write(indent),
      vis_body1(Body).

vis_term(var(V)) :- str_int(V,I), write("V",I), !.
vis_term(var(V)) :- write(V),!.
vis_term(int(C)) :- write(C),!.
vis_term(str(C)) :- write("\"",C,"\""),!.
vis_term(plus(E1,E2)) :- vis_term(E1), write(" + "), vis_term(E2),!.
vis_term(minus(E1,E2)) :- vis_term(E1), write(" - "), vis_term(E2),!.

vis_term(mult(E1,E2))
   :- vis_ass_term(E1), write(" * "), vis_ass_term(E2),!.
vis_term(div(E1,E2))
   :- vis_ass_term(E1), write(" / "), vis_ass_term(E2),!.

vis_term(anonymousvariable) :- write("_"),!.
vis_term(sign(E)) :- write("-"), vis_term(E),!.
vis_term(q(1,N,1)) :- write(N),!.
vis_term(q(1,N,D)) :- write(N,"/",D),!.
vis_term(q(-1,N,1)) :- write("(-",N,")"),!.
vis_term(q(-1,N,D)) :- write("(-",N,"/",D,")"),!.
vis_term(anonymousvariable()) :- write("_"), !.
vis_term(Expr) :- write(Expr).


vis_ass_term(plus(E1,E2))
   :- write("("), vis_term(plus(E1,E2)), write(")"),!.
vis_ass_term(minus(E1,E2))
   :- write("("), vis_term(minus(E1,E2)), write(")"),!.
vis_ass_term(sign(E))
   :- write("("), vis_term(sign(E)), write(")"),!.
vis_ass_term(Expr)
   :- vis_term(Expr).
```

```
vis_literal(pos(Atom)) :- vis_atom(Atom).
vis_literal(neg(Atom)) :- write("not("), vis_atom(Atom),
                          write(")").

vis_atom(head(ID,TS))
   :- write(ID), vis_termlist(TS), !.
vis_atom(less_than(E1,E2))
   :- vis_term(E1), write(" < "), vis_term(E2),!.
vis_atom(less_equal(E1,E2))
   :- vis_term(E1), write(" <= "), vis_term(E2),!.
vis_atom(greater_than(E1,E2))
   :- vis_term(E1), write(" > "), vis_term(E2),!.
vis_atom(greater_equal(E1,E2))
   :- vis_term(E1), write(" >= "), vis_term(E2),!.
vis_atom(not_equal(E1,E2))
   :- vis_term(E1), write(" != "), vis_term(E2),!.
vis_atom(equal(E1,E2))
   :- vis_term(E1), write(" = "), vis_term(E2),!.
vis_atom(Atom) :- write(Atom).

vis_query(query(pos(head(ID,QTermlist))))
   :- write("?- ",ID), vis_qtermlist(QTermlist), write(".").

vis_qtermlist([]) :- !.
vis_qtermlist([T|TS])
   :- write("("), vis_qterm(T), vis_qtermlist1(TS), write(")").

vis_qtermlist1([]) :- !.
vis_qtermlist1([T|TS])
   :- write(","), vis_qterm(T), vis_qtermlist1(TS).

vis_qterm(var(V)) :- write(V), !.
vis_qterm(signrat(A,B)) :- write("-",A,"/",B), !.
vis_qterm(rat(A,B)) :- write(A,"/",B), !.
vis_qterm(sign(I)) :- write("-",I), !.
vis_qterm(int(I)) :- write(I), !.
vis_qterm(anonymousvariable()) :- write("_"), !.
vis_qterm(str(S)) :- write(S).
```

G.6 MODULE fortolk.pro

```
project "datalog"
include "globals.pro"
include "settools.pro"
include "optimize.pro"


PREDICATES

  rulegraph_to_db       (Program)
  transform             (string,Dbase,Dbase)
  optimize              (Query,Dbase,Program)
  intro_av              (Program,Program)

  database_depend       (Dbase,Depends)
  insert_rule           (Rule,Depends,Depends)
  insert_depend         (Depend,Depends,Depends)
  body_depend           (Body,Strings)

  depends_graf          (Depends,DependGraf)

  graf1                 (Depends,DependGraf,DependGraf,Depends,integer)
  graf2                 (Depends,DependGraf,DependGraf)

  test_forrige          (Depend,DependGraf)
  test_forrige1         (Strings,DependGraf)
  rule_with             (STRING,DependGraf)
  init_graf             (Depends,Depends,DependGraf,Depends)
  init_graf1            (Depend,Depends,Depends,DependGraf)
  init_graf2            (Strings,Depends,Depends,DependGraf)
  find_gruppe           (Depends,Depends,DependGraf,Depends,Depends)
  nste_depend          (Depends,DependGraf,Depend)
  split_depend          (Depends,Dependgraf,Depends,Depends)
  test_reach            (Depends,Depends,Depend)
  reachable             (Depend,Depend,Depends)

  test_forrige2         (Strings,DependGraf)

  endnu_ikke_placeret   (Strings,DependGraf,Strings)
  udtag_regler          (Strings,Depends,Depends,Depends)
  udtag_regel           (String,Depends,Depend,Depends)

  computation_graf      (Query,DependGraf,DependGraf)
  rules_needed          (Query,DependGraf,Strings,DependGraf)
  case_need             (Strings,Depends,Strings,DependGraf)
  names                 (Depends,Strings)
  depends               (Depends,Strings)

facts_to_edb            (Dbase,Dbase)
   fact                 (Rule)
   fact2                (Termlist)

  make_plan             (Dbase,DependGraf,DependGraf,Program)
     case_plan          (Dbase,Program)
  make_plan2            (Dbase,Depends,DependGraf,Program,Dbase,Dbase)
     case_plan2         (Dbase,Program)
  split_rules           (STRING,Dbase,Dbase,Dbase)
  split_Srules          (Dbase,DependGraf,Dbase,Dbase)

optimize_recursive      (Codes,Codes,Program,Program)
```

```
optimize_unfolding      (Codes,Codes,Program,Program)
optimize_program        (Codes,Codes,Program,Program)
optimize_rules          (Codes,Codes,Dbase,Dbase)
optimize_rule           (Codes,Codes,Rule,Rule)
   equal_rule           (Rule,Rule)
   optimize_numeric     (Codes,Codes,Rule,Rule)
   optimize_negcons     (Codes,Codes,Rule,Rule)
      negcon2pos        (Codes,Body,Body)
         negcon2pos2    (Codes,Literal,Literal)
   optimize_equal       (Codes,Codes,Rule,Rule)
      find_eq_rule      (Rule,SubstSet,Rule)
      find_eq_body      (Body,SubstSet,Body)
      find_eq_literal   (Literal,SubstSet,Body)
      find_eq_atom      (Atom,SubstSet,Body)

optimize_fail           (Codes,Codes,Program,Program)

intro_av_db             (Dbase,Dbase)
intro_av_rule           (Rule,Rule)
preparesub              (OccurSet,SubstSet)
count_rule              (Rule,OccurSet)
count_body              (Body,OccurSet,OccurSet)
count_literal           (Literal,OccurSet,OccurSet)
count_atom              (Atom,OccurSet,OccurSet)
count_head              (Head,OccurSet,OccurSet)
count_expr              (Expr,OccurSet,OccurSet)
count_termlist          (Termlist,OccurSet,OccurSet)

insert_map              (Expr,OccurSet,OccurSet)

subst_set               (SubstSet,Rule,Rule)

substitute              (Expr,Expr,Rule,Rule)
sub_body                (Expr,Expr,Body,Body)
sub_literal             (Expr,Expr,Literal,Literal)
sub_atom                (Expr,Expr,Atom,Atom)
sub_head                (Expr,Expr,Head,Head)
sub_termlist            (Expr,Expr,Termlist,Termlist)
sub_expr                (Expr,Expr,Expr,Expr)



    present         (relation)
    present1        (Termlist,forest,integer)
    present2        (Termlist,tree,integer)

  pick_tree        (forest,tree,forest)
```

```
CONSTANTS
  preline = "\n-- "
  postline =" -----------------------------------\n"

  warning1= "INIT"
%  warning2= "Fejl: Kunne ikke finde nogen lsning."
%  warning3= "Fejl: ",S," reglerne har forskellig grad."
%  warning4= "Fejl: Reglen ",S," kan ikke beregnes."


%  verbosity = 1


CLAUSES
/* include interpreter from here -----------------*/

interpret(DB,Query) :-
        file_str(outname,FatalErrorMessages),

        openwrite(outfile,outname),
        writedevice(outfile),

        write("-- ","MESSAGES",postline),nl,
        write(FatalErrorMessages),nl,

        source(Tekst),!,

        write(preline,"SOURCE TEXT",postline),
        write(Tekst),

ifdef verbosity
        write(preline,"QUERY",postline),
%        vis_query(Query),nl,
        write(Query), nl,

%        write(preline,"DATABASE",postline),
%        vis_db(DB),
%        vis_result2(DB),
enddef

        transform(FatalErrorMessages,DB,DB1),

        optimize(Query,DB1,Program1),
        intro_av(Program1,Program2),

ifdef verbosity
        write(preline,"OPTIMIZED PROGRAM",postline),
        vis_resultat3(Program2),

%        adorn(Program2),
        write(preline,"ADORNMENTS - BINDING PATTERNS",postline),
        vis_all_adornment,!,
        write(preline,"SAFETY INFORMATION",postline),
        vis_all_safety,!,
        write(preline,"MANUALLY RANGE-RESTRICTION INFORMATION",postline),
        vis_all_range,!,
        write(preline,"INTERPRETER RECURSIVE QUERY SUBQUERY",postline),
enddef


        rulegraph_to_db(Program2),
        interpret_program(Query,Answer),

        write(preline,"THE ANSWER TO THE QUERY",postline),
        present(Answer),

        closefile(outfile),

        file_str(outname,OUTSTRING),

        % SAVE LOG FILE
        filename(FullName),
        filenameext(FullName,Name,_),
        filenameext(OutName,Name,".LOG"),
        file_str(OutName,OUTSTRING),

        edit(OUTSTRING,_,"",outname,"Press F10 to return to menu.",
        0,"DATALOG.HLP",1,1,1,0,_,_), !.
interpret(_,_) :-
        write("ERROR: Interpreter failed"),nl,
        closefile(outfile),
        file_str(outname,OUTSTRING),
        edit(OUTSTRING,_,"",outname,"Press F10 to return to menu.",
        0,"DATALOG.HLP",1,1,1,0,_,_), !.


  present (r(_,I,S)) :- present1(I,S,0),nl.

  present1(_,[],0)        :- write("NO"), nl,                 !.
  present1(_,[leaf],0)        :- write("YES"),  nl,                !.
  present1(_,[],1)        :- nl, write("One solution."), nl,         !.
  present1(_,[],L)        :- nl, write(L," solutions."), nl,        !.
  present1(I, S,L)
     :- pick_tree (S,Solution,R),
        nl, present2 (I,Solution,C), L1 = L+C,
        present1(I,R,L1).

  present2 ([X],    t(anonymousvariable,[leaf]),1)
     :- vis_term(X), write("=_."), !.
  present2 ([X|R],  t(anonymousvariable,[T]),I)
     :- vis_term(X), write("=_, "), present2(R,T,I), !.
  present2 ([X],         t(X,[leaf]),1)
```

```
     :- write("."), !.
  present2 ([X|R],       t(X,[T]),I)
     :- present2(R,T,I), !.
  present2 ([X],         t(Y,[leaf]),1)
     :- vis_term(X), write("="), vis_term(Y), write("."), !.
  present2 ([X|R],       t(Y,[T]),I)
     :- vis_term(X), write("="), vis_term(Y), write(", "),
        present2(R,T,I), !.
  present2 ([_|R],        t(_,[T]),I)
     :- present2(R,T,I), !.


  pick_tree ([leaf],leaf,[]) :- !.
  pick_tree ([t(A,S)|R],t(A,[T]),R)
     :- pick_tree(S,T,[]), !.
  pick_tree ([t(A,S)|R],t(A,[T]),[t(A,R1)|R])
     :- pick_tree(S,T,R1), !.



rulegraph_to_db([repeat(Rules1)|R1])
   :- facts_to_edb(Rules1,Rules2),
      assert(rulegraph_db(repeat(Rules2))),
      rulegraph_to_db(R1),!.
rulegraph_to_db([init(S)|R1])
   :- assert(rulegraph_db(init(S))),
      rulegraph_to_db(R1),!.
rulegraph_to_db([step(Rules1)|R1])
   :- facts_to_edb(Rules1,Rules2),
      assert(rulegraph_db(step(Rules2))),
      rulegraph_to_db(R1),!.
rulegraph_to_db([]).







transform(FatalErrorMessages,_,[])
   :- searchstring(FatalErrorMessages,"FATAL ERROR:",_), !.
transform(_,DB,DB1)
   :- t_database(DB,DB1).
%      write(preline,"Rewriting Expressions",postline),
%      vis_db(DB1).


optimize(_,[],[]) :- !.
optimize(Query,DB,Program2)
   :-        database_depend(DB,Depends),
%        write(preline,"Rules depends of",postline),
%        vis_resultat2(Depends),

        depends_graf(Depends,Graf),
%        write(preline,"Graph of dependencies",postline),
%        vis_resultat2(Graf),

        computation_graf(Query,Graf,Graf2),
%        write(preline,"Graph of relevant dependencies",postline),
%        vis_resultat2(Graf2),

        facts_to_edb(DB,IDB),
%        write(preline,"INTENSIONAL DATABASE",postline),
%        vis_db(IDB),
%        write(preline,"EXTENSIONAL DATABASE",postline),
%        vis_all_facts,!,

        make_plan(IDB,[],Graf2,Program1),
%        write(preline,"Rule graph",postline),
%        vis_resultat3(Program1),

        optimize_recursive([unfold,intro_equal],_,Program1,Program2),
/*        write(preline,"Intermediate optimized rule graph",postline),
        vis_resultat3(Program2),
        write(preline,"Codes",postline),
        vis_resultat2(CodesIntroduced),*/ !.
optimize(_,_,[])
   :- write(preline,"ERROR: optimize predicate failed", postline).


database_depend([],[]) :- !.
database_depend([Rule|Rest],Depend2)
   :- database_depend(Rest,Depend1),
      insert_rule(Rule,Depend1,Depend2).

insert_rule(rule(head(S,_),DL),D1,D2)
   :- body_depend(DL,Depend),
      insert_depend(a(S,Depend),D1,D2), !.
insert_rule(_,D,D). % Otherwise range and safety

insert_depend(A,[],[A]) :- !.
insert_depend(a(S,DL1),[a(S,DL2)|Rest],[a(S,DL3)|Rest])
   :- union(DL1,DL2,DL3), !.
insert_depend(A,[B|Rest],[B|Depend])
   :- insert_depend(A,Rest,Depend), !.

body_depend([],[]) :- !.
body_depend([pos(head(S,_))|R],Depend2)
   :- body_depend(R,Depend1),
      union([S],Depend1,Depend2), !.
body_depend([neg(head(S,_))|R],Depend2)
   :- body_depend(R,Depend1),
      union([S],Depend1,Depend2), !.
body_depend([_|R],Depend)
   :- body_depend(R,Depend), !. % Otherwise constraints


depends_graf(AL,G)
   :- init_graf(AL,AL,IG,AL1),
      card(AL1,N),
      graf1(AL1,IG,G,[],N),!.

/* ny version, virker tilsyneladende efter test */
graf1([A|R],G1,G3,A_vent,N1)
   :- test_forrige(A,G1),!,
      append(G1,[[A]],G2),!,
      N=N1-1,
      graf1(R,G2,G3,A_vent,N), !.

/* forsg p forbedring, gl version
graf1([A|R],G1,G3,A_vent,N)
   :- test_forrige(A,G1),!,
      append(G1,[[A]],G2),!,
      graf1(R,G2,G3,A_vent,N), !.
*/
graf1([A|R],G1,G2,A_vent,N)
   :- graf1(R,G1,G2,[A|A_vent],N), !.

graf1([],G1,G2,AL,N)
   :- N<1, graf2(AL,G1,G2), !.

graf1([],G1,G2,AL,N1)
   :- N=N1-1, graf1(AL,G1,G2,[],N), !.

graf2([],G,G) :- !.
graf2(AL,G1,G3)
   :- nste_depend(AL,G1,A),
      remove(A,AL,AL2),
      find_gruppe([A],AL2,G1,Gruppe,R2),
      append(G1,[Gruppe],G2),
      card(R2,N),
      graf1(R2,G2,G3,[],N), !.

% test_forrige tester at reglen S kun afhnger af enten
% sig selv eller allerede placerede regler.

test_forrige(a(S,D),G)
   :- remove(S,D,D1), test_forrige1(D1,G).

   % test_forrige1 tester at man kun afhnger af Graf.
   test_forrige1([],_) :- !.
   test_forrige1([D|R],G)
      :- rule_with(D,G), test_forrige1(R,G).
      rule_with(D,[[a(D,_)|_]|_])        :- !.
      rule_with(D,[[_|L]|_])                :- member(a(D,_),L), !.
      rule_with(D,[_|R])                :- rule_with(D,R), !.

/* grafen initialiseres med alle facts */
init_graf(_,[],[],[]) :- !.
init_graf(AL,[A|R],[[A]|G],R1)
   :- A = a(_,[]), init_graf(AL,R,G,R1), !.

init_graf(AL,[A|R],G,[A|R1])
   :- init_graf1(A,AL,AL2,G1), init_graf(AL2,R,G2,R1),
      append(G1,G2,G), !.


% Nr underml ikke findes som regel eller facts.
   init_graf1(a(_,L),R,R2,G)
      :- init_graf2(L,R,R2,G).

   init_graf2([],R,R,[]) :- !.
   init_graf2([S|Rest],R,R2,G)
      :- member(a(S,_),R),
         init_graf2(Rest,R,R2,G), !.
   init_graf2([S|Rest],R,R2,[[a(S,[warning1])]|G])
      :- init_graf2(Rest,[a(S,[])|R],R2,G),!.


% Nr nste_depend kaldes, vides med sikkerhed at der
% findes en cyklus i grafen. I denne cyklus vil der vre
% mindst en regel der er afhngig af fortiden. Udfra denne
% regel skal man finde gruppen.


nste_depend(AL,Graf,A)
   :- split_depend(AL,Graf,AL1,AL2),
trace(on),      test_reach(AL1,AL2,A), !.

   % split_depend opdeler AL i to dele, AL1 som afhnger af regler i
   % GRAF og AL2 som ikke afhnger af noget kendt.


   split_depend([],_,[],[]) :- !.
   split_depend([A|R1],Graf,[A|R2],AL)
      :- A = a(S,DL),
         remove(S,DL,DL1),
         test_forrige2(DL1,Graf),
         split_depend(R1,Graf,R2,AL), !.
   split_depend([A|R1],Graf,R2,[A|AL])
      :- split_depend(R1,Graf,R2,AL), !.

   % test_reach udvlger det frste A der kan n sig selv.
   test_reach([A|R],AL,A) :- append(R,AL,AL2), reachable(A,A,AL2), !.
   test_reach([_|R],AL,A) :- test_reach(R,AL,A), !.


   test_reach([],[A|R],A) :- reachable(A,A,R), !.
   test_reach([],[_|R],A) :- test_reach([],R,A), !.






   % reachable kontrolere at X er forbundet med Y.

   reachable(X,Y,_)
      :- not (X=Y),
         Y = a(Sy,_),
         X = a(_,DLx),
         member(Sy,DLx),!.

   reachable(X,Y,L)
      :- X = a(_,DLx),
         member(Sz,DLx),
         member(a(Sz,DLz),L),
         remove(a(Sz,DLz),L,L1),
         reachable(a(Sz,DLz),Y,L1), !.

   % test_forrige2 tester at man afhnger af Graf.
   test_forrige2([D|_],Graf)
      :- rule_with(D,Graf), !.
   test_forrige2([_|R],Graf)
      :- test_forrige2(R,Graf), !.


% find_gruppe finder de regler, der er gensidigt afhngige af A

find_gruppe([],Rest,_,[],Rest) :- !.
find_gruppe([A|R1],Rest1,Graf,[A|Gruppe],Rest3)  /*(i,i,i,o,o)*/
   :- A = a(S,DL1),
      remove(S,DL1,DL2),
      endnu_ikke_placeret(DL2,Graf,DL3),
      udtag_regler(DL3,Rest1,R2,Rest2),
      append(R1,R2,R3),
      append(Graf,[[A]],Graf2),
      find_gruppe(R3,Rest2,Graf2,Gruppe,Rest3), !.


% endnu_ikke_placeret: af en mngde navne findes dem som ikke er
% placeret i grafen endnu.

endnu_ikke_placeret([],_,[]) :-!.
endnu_ikke_placeret([D|R1],Graf,R2)
   :- rule_with(D,Graf), endnu_ikke_placeret(R1,Graf,R2), !.
endnu_ikke_placeret([D|R1],Graf,[D|R2])
   :- endnu_ikke_placeret(R1,Graf,R2), !.

% Depend opdeles i dem som er med i Listen og dem som ikke er med.
% Hvis der ikke findes et Depend til Navnet udskrives en fejlmeddelelse.

udtag_regler([],Rest,[],Rest) :- !.
udtag_regler([Navn|Rest],Depend1,[A|AL],Depend3)
   :- udtag_regel(Navn,Depend1,A,Depend2),
      udtag_regler(Rest,Depend2,AL,Depend3), !.

udtag_regel(Navn,[a(Navn,DL)|R],a(Navn,DL),R) :- !.
udtag_regel(Navn,[B|R1],A,[B|R2])
   :- udtag_regel(Navn,R1,A,R2).


computation_graf(Query,G1,G2)
   :- rules_needed(Query,G1,_,G2), !.

computation_graf(_,_,[]). % Fejl hvis det naar her til.


rules_needed(Query,[Group1|Rest],Set2,Graf3)
   :- rules_needed(Query,Rest,Set1,Graf),
      case_need(Set1,Group1,Set2,Graf2),
      append(Graf2,Graf,Graf3), !.

rules_needed(query(pos(head(Query,_))),[],[Query],[]).


case_need(Set1,Group,Set4,[Group])
   :- names(Group,Set2),
      member(M,Set2),
      member(M,Set1),
      depends(Group,Set3),
      union(Set1,Set3,Set4), !.

case_need(Set,_,Set,[]).


names([],[]) :-!.
names([a(S,_)|R],[S|NS])
   :- names(R,NS).

depends([],[]) :-!.
depends([a(_,DL)|R],NS2)
   :- depends(R,NS1), union(DL,NS1,NS2).
```

```
facts_to_edb([],[]) :- !.
facts_to_edb([Rule|DB],IDB)
   :- fact(Rule),          % Write facts to the external database.
      Rule = rule(head(ID,TS),_),
      assert(fact_edb(ID,TS)),
      facts_to_edb(DB,IDB),!.
facts_to_edb([VirtualRule|DB],[VirtualRule|IDB])
   :- facts_to_edb(DB,IDB), !.

   fact(rule(head(_,TS),[])) :- fact2(TS).
   fact2([]) :- !.
   fact2([q(_,_,_)|TS]) :- fact2(TS), !.
   fact2([str(_)|TS]) :- fact2(TS), !.


  make_plan(Rules,Bottom,[ [a(S,[warning1])] |Top],[init(S)|Program])
     :- make_plan(Rules,[[a(S,[warning1])]|Bottom],Top,Program), !.
  make_plan(Rules1,Bottom1,[Group|Top],Program)
     :- make_plan2(Rules1,Group,Bottom1,Program1,Repeats,Rules2),
        case_plan(Repeats,Program2),
        append(Bottom1,[Group],Bottom2),
        make_plan(Rules2,Bottom2,Top,Program3),
        append(Program1,Program2,Temp),
        append(Temp,Program3,Program), !.
  make_plan(_,_,[],[]).

     case_plan([],[]) :- !.
     case_plan(R,[repeat(R)]).

  make_plan2(Rules1,[a(S,_)|RestGroup],Bottom,Steps,Repeats,Rules3)
     :- split_rules(S,Rules1,SRules,Rules2),
        split_Srules(SRules,Bottom,RuleStep,Repeat1),
        case_plan2(RuleStep,Step1),
        make_plan2(Rules2,RestGroup,Bottom,Step2,Repeat2,Rules3),
        append(Step1,Step2,Steps),
        append(Repeat1,Repeat2,Repeats), !.
  make_plan2(Rules,[],_,[],[],Rules).

     case_plan2([],[]) :- !.
     case_plan2(S,[step(S)]).

  split_rules(S,[R|Rules1],[R|SRules],Rules2)
     :- R = rule(head(S,_),_),
        split_rules(S,Rules1,SRules,Rules2), !.
  split_rules(S,[R|Rules1],SRules,[R|Rules2])
     :- split_rules(S,Rules1,SRules,Rules2), !.
  split_rules(_,[],[],[]).

  split_Srules([R|Rules],Bottom,[R|Steps],Repeats)
     :- R = rule(_,Literals),
        body_depend(Literals,Depends),
        test_forrige1(Depends,Bottom),
        split_Srules(Rules,Bottom,Steps,Repeats), !.
  split_Srules([R|Rules],Bottom,Steps,[R|Repeats])
     :- split_Srules(Rules,Bottom,Steps,Repeats), !.
  split_Srules([],_,[],[]).



optimize_recursive(Codes1,Codes4,Program1,Program4)
   :- optimize_unfolding(Codes1,Codes2,Program1,Program2),
      optimize_program(Codes2,Codes3,Program2,Program3),
      optimize_fail(Codes3,Codes4,Program3,Program4), !.
optimize_recursive(Codes,Codes,Program,Program)
   :- write(preline,"ERROR: optimize recursive predicate failed",
      postline).

optimize_unfolding(Codes,Codes,Program,Program).

optimize_program(Codes,Codes,[],[]) :- !.
optimize_program(Codes1,Codes3,[step(Rule1)|Program1],
                 [step(Rule2)|Program2])
   :- optimize_rules(Codes1,Codes2,Rule1,Rule2),
      optimize_program(Codes2,Codes3,Program1,Program2), !.
optimize_program(Codes1,Codes3,[repeat(Rule1)|Program1],
                 [repeat(Rule2)|Program2])
   :- optimize_rules(Codes1,Codes2,Rule1,Rule2),
      optimize_program(Codes2,Codes3,Program1,Program2), !.
optimize_program(Codes1,Codes2,[init(S)|Program1],[init(S)|Program2])
   :- optimize_program(Codes1,Codes2,Program1,Program2), !.

optimize_rules(Codes,Codes,[],[]) :- !.
optimize_rules(Codes1,Codes3,[Rule1|Program1],[Rule2|Program2])
   :- optimize_rule(Codes1,Codes2,Rule1,Rule2),
      optimize_rules(Codes2,Codes3,Program1,Program2).

optimize_rule(Codes1,Codes5,Rule1,Rule5)
   :- optimize_numeric(Codes1,Codes2,Rule1,Rule2),
      optimize_negcons(Codes2,Codes3,Rule2,Rule3),
      optimize_equal(Codes3,Codes4,Rule3,Rule4),
      not (equal_rule(Rule4,Rule1)), !,
      optimize_rule(Codes4,Codes5,Rule4,Rule5), !.
optimize_rule(Codes,Codes,Rule,Rule).

  equal_rule(rule(head(ID,TS),Body1),rule(head(ID,TS),Body2))
     :- equal_set(Body1,Body2).


   optimize_numeric(Codes,Codes,Rule,Rule).

   optimize_negcons(CodesIn,CodesOut,
                           rule(head(ID,TS),Body1),
                           rule(head(ID,TS),Body2))
      :- member(unfold,CodesIn), !,
         negcon2pos(CodeOut,Body1,Body2),
         union(CodeOut,CodesIn,CodesOut),!.
   optimize_negcons(Codes,Codes,Rule,Rule).
   %Only optimize when unfolding has been done.

      negcon2pos(CodeOut3,[L1|Body1],Body3)
         :- negcon2pos2(CodeOut1,L1,L2),
            negcon2pos(CodeOut2,Body1,Body2),
            insert(L2,Body2,Body3),
            union(CodeOut1,CodeOut2,CodeOut3), !.
      negcon2pos([],[],[]) :- !.

         negcon2pos2([],pos(Atom),
         pos(Atom)) :- !.
         negcon2pos2([],neg(head(ID,TS)),
         neg(head(ID,TS))) :- !.
         negcon2pos2([],neg(equal(Expr1,Expr2)),
         pos(not_equal(Expr1,Expr2))) :- !.
         negcon2pos2([],neg(less_than(Expr1,Expr2)),
         pos(less_equal(Expr2,Expr1))) :- !.
         negcon2pos2([],neg(less_equal(Expr1,Expr2)),
         pos(less_than(Expr2,Expr1))) :- !.
         negcon2pos2([intro_equal],neg(not_equal(Expr1,Expr2)),
         pos(equal(Expr1,Expr2))) :- !.



   optimize_equal(Codes,Codes,Rule1,Rule3)
      :- member(intro_equal,Codes), !,
         find_eq_rule(Rule1,Set,Rule2),
         subst_set(Set,Rule2,Rule3), !.
   optimize_equal(Codes,Codes,Rule,Rule)
      :- write("ERROR: optimize equal predicate failed").


      find_eq_rule(rule(Head,Body1),Set,rule(Head,Body2))
         :- find_eq_body(Body1,Set,Body2).

      find_eq_body([],[],[]) :- !.
      find_eq_body([L|Body1],Set,Body3)
         :- find_eq_literal(L,Set,Body2),
            union(Body2,Body1,Body3), !.
      find_eq_body([L|Body1],Set,Body3)
         :- find_eq_body(Body1,Set,Body2),
            insert(L,Body2,Body3).

      find_eq_literal(pos(Atom),Set,Body)
         :- find_eq_atom(Atom,Set,Body), !.
      find_eq_atom(equal(var(X),var(Y)),
                      [sub(var(X),var(Y))],[]) :- !.
      find_eq_atom(equal(var(X),q(S,N,D)),
                      [sub(var(X),q(S,N,D))],[]) :- !.
      find_eq_atom(equal(var(X),str(Y)),
                      [sub(var(X),str(Y))],[]) :- !.
      find_eq_atom(equal(q(S,N,D),var(Y)),
                      [sub(var(Y),q(S,N,D))],[]) :- !.
      find_eq_atom(equal(str(X),var(Y)),
                      [sub(var(Y),str(X))],[]) :- !.
      find_eq_atom(equal(q(S,N,D),q(S,N,D)),[],[]) :- !.
      find_eq_atom(equal(str(X),str(X)),[],[]) :- !.
      find_eq_atom(equal(q(_,_,_),q(_,_,_)),[],
                      [pos(head("fail",[]))]) :- !.
      find_eq_atom(equal(str(_),str(_)),[],
                      [pos(head("fail",[]))]) :- !.
      find_eq_atom(equal(anonymousvariable,_),[],[]) :- !.
      find_eq_atom(equal(_,anonymousvariable),[],[]) :- !.

% In case the type check is down, then this will not fail.
      find_eq_atom(equal(q(_,_,_),str(_)),[],
                      [pos(head("type error",[]))]) :- !.
      find_eq_atom(equal(str(_),q(_,_,_)),[],
                      [pos(head("type error",[]))]) :- !.



optimize_fail(Codes,Codes,Program,Program).


intro_av([],[]) :- !.
intro_av([step(Rule1)|Program1],[step(Rule2)|Program2])
   :- intro_av_db(Rule1,Rule2),
      intro_av(Program1,Program2), !.
intro_av([repeat(Rule1)|Program1],[repeat(Rule2)|Program2])
   :- intro_av_db(Rule1,Rule2),
      intro_av(Program1,Program2), !.
intro_av([init(S)|Program1],[init(S)|Program2])
   :- intro_av(Program1,Program2), !.

intro_av_db([],[]) :- !.
intro_av_db([Rule1|DB1],[Rule2|DB2])
   :- intro_av_rule(Rule1,Rule2),
      intro_av_db(DB1,DB2).

intro_av_rule(rule(Head,Body),Rule)
   :- count_rule(rule(Head,Body),Map),
      preparesub(Map,Set),
      subst_set(Set,rule(Head,Body),Rule).

preparesub([],[]) :- !.
preparesub([occur(Var,1)|Rest],[sub(Var,anonymousvariable)|Set])
   :- preparesub(Rest,Set), !.
preparesub([_|Rest],Set)
   :- preparesub(Rest,Set).


count_rule(rule(Head,Body),Map)
   :- count_head(Head,[],Map1),
      count_body(Body,Map1,Map).

count_body([],Map,Map) :- !.
count_body([L|Body],Map1,Map)
   :- count_literal(L,Map1,Map2),
      count_body(Body,Map2,Map).

count_literal(pos(Atom),Map1,Map2)
   :- count_atom(Atom,Map1,Map2), !.
count_literal(neg(Atom),Map1,Map2)
   :- count_atom(Atom,Map1,Map2), !.

count_head(head(_,TS),Map1,Map2)
   :- count_termlist(TS,Map1,Map2), !.

count_atom(head(_,TS),Map1,Map2)
   :- count_termlist(TS,Map1,Map2), !.
count_atom(less_equal(T1,T2),Map1,Map3)
   :- count_expr(T1,Map1,Map2),
      count_expr(T2,Map2,Map3).
count_atom(less_than(T1,T2),Map1,Map3)
   :- count_expr(T1,Map1,Map2),
      count_expr(T2,Map2,Map3).
count_atom(not_equal(T1,T2),Map1,Map3)
   :- count_expr(T1,Map1,Map2),
      count_expr(T2,Map2,Map3).
count_atom(equal(E1,T2),Map1,Map3)
   :- count_expr(E1,Map1,Map2),
      count_expr(T2,Map2,Map3).

count_expr(plus(T1,T2),Map1,Map3)
   :- count_expr(T1,Map1,Map2),
      count_expr(T2,Map2,Map3),!.
count_expr(mult(T1,T2),Map1,Map3)
   :- count_expr(T1,Map1,Map2),
      count_expr(T2,Map2,Map3),!.
count_expr(sign(T),Map1,Map2)
   :- count_expr(T,Map1,Map2),!.
count_expr(var(V),Map1,Map2)
   :- insert_map(var(V),Map1,Map2), !.
count_expr(_,Map,Map).

count_termlist([],Map,Map) :- !.
count_termlist([T|TS],Map1,Map3)
   :- count_expr(T,Map1,Map2),
      count_termlist(TS,Map2,Map3).



insert_map(Var,Map1,Map3)
   :- member(occur(Var,Count),Map1), !,
      remove(occur(Var,Count),Map1,Map2),
      Count1 = Count + 1,
      append([occur(Var,Count1)],Map2,Map3), !.
insert_map(Var,Map1,Map2)
   :- append([occur(Var,1)],Map1,Map2), !.



subst_set([],Rule,Rule) :- !.
subst_set([sub(X,Y)|Rest],Rule1,Rule3)
   :- substitute(X,Y,Rule1,Rule2),
      subst_set(Rest,Rule2,Rule3).

substitute(X,Y,rule(Head1,Body1),rule(Head2,Body2))
   :- sub_head(X,Y,Head1,Head2),
      sub_body(X,Y,Body1,Body2).

sub_body(_,_,[],[]) :- !.
sub_body(X,Y,[L1|Body1],Body3)
   :- sub_literal(X,Y,L1,L2),
      sub_body(X,Y,Body1,Body2),
      insert(L2,Body2,Body3).

sub_literal(X,Y,pos(Atom1),pos(Atom2))
   :- sub_atom(X,Y,Atom1,Atom2), !.
sub_literal(X,Y,neg(Atom1),pos(Atom2))
   :- sub_atom(X,Y,Atom1,Atom2), !.

sub_head(X,Y,head(ID,TS1),head(ID,TS2))
   :- sub_termlist(X,Y,TS1,TS2), !.

sub_atom(X,Y,head(ID,TS1),head(ID,TS2))
   :- sub_termlist(X,Y,TS1,TS2), !.
sub_atom(X,Y,less_than(T1,T2),less_than(T3,T4))
   :- sub_expr(X,Y,T1,T3),
      sub_expr(X,Y,T2,T4),!.
sub_atom(X,Y,less_equal(T1,T2),less_equal(T3,T4))
   :- sub_expr(X,Y,T1,T3),
      sub_expr(X,Y,T2,T4),!.
sub_atom(X,Y,not_equal(T1,T2),not_equal(T3,T4))
   :- sub_expr(X,Y,T1,T3),
      sub_expr(X,Y,T2,T4),!.
sub_atom(X,Y,equal(E1,T1),equal(E2,T2))
```

```
   :- sub_expr(X,Y,E1,E2),
      sub_expr(X,Y,T1,T2),!.

sub_termlist(_,_,[],[]) :- !.
sub_termlist(X,Y,[T1|TS1],[T2|TS2])
   :- sub_expr(X,Y,T1,T2),
      sub_termlist(X,Y,TS1,TS2).

sub_expr(X,Y,plus(T1,T2),plus(T3,T4))
   :- sub_expr(X,Y,T1,T3),
      sub_expr(X,Y,T2,T4), !.
sub_expr(X,Y,mult(T1,T2),mult(T3,T4))
   :- sub_expr(X,Y,T1,T3),
      sub_expr(X,Y,T2,T4), !.
sub_expr(X,Y,sign(T1),sign(T2))
   :- sub_expr(X,Y,T1,T2), !.
sub_expr(X,Y,X,Y) :- !.
sub_expr(_,_,T,T).
```

G.6.1 INCLUDE optimize.pro

```
PREDICATES

t_database        (Dbase,Dbase)
t_rule                (Rule,Rule)
t_body                (Body,integer,Body,integer)
t_literal        (Literal,integer,Body,integer)
t_atom                (Atom,integer,Body,Atom,integer)
t_head                (Head,integer,Body,Head,integer)
t_head                (Atom,integer,Body,Atom,integer)
t_termlist        (Termlist,integer,Body,Termlist,integer)
t_built_in        (Atom,integer,Body,Atom,integer)
t_expr                (Expr,integer,Body,Expr,integer)


CLAUSES


t_database([],[]) :- !.
t_database([Rule1|DB1],DB3)
   :- t_rule(Rule1,Rule2),
      t_database(DB1,DB2),
      insert(Rule2,DB2,DB3).

t_rule(rule(Head,Body),rule(Head1,Body3))
   :- t_body(Body,0,Body1,Var1),
      t_head(Head,Var1,Body2,Head1,_),
      union(Body1,Body2,Body3), !.
t_rule(Range,Range).
```

```
t_body([],Var,[],Var) :- !.
t_body([L|Body],Var,Body3,Var2)
   :- t_literal(L,Var,Body1,Var1),
      t_body(Body,Var1,Body2,Var2),
      union(Body1,Body2,Body3).

t_literal(pos(Atom),Var,Body1,Var1)
   :- t_atom(Atom,Var,Body,Atom1,Var1),
      insert(pos(Atom1),Body,Body1), !.
t_literal(neg(Atom),Var,Body1,Var1)
   :- t_atom(Atom,Var,Body,Atom1,Var1),
      insert(neg(Atom1),Body,Body1), !.

t_atom(head(P,TS),Var,Body,Atom1,Var1)
   :- t_head(head(P,TS),Var,Body,Atom1,Var1), !.
t_atom(Atom,Var,Body,Atom1,Var1)
   :- t_built_in(Atom,Var,Body,Atom1,Var1).

t_head(head(P,TS),Var,Body,head(P,TS1),Var1)
   :- t_termlist(TS,Var,Body,TS1,Var1).

t_termlist([],Var,[],[],Var) :- !.
t_termlist([T|TS],Var,Body3,TS2,Var2)
   :- t_termlist(TS,Var,Body1,TS1,Var1),
      t_expr(T,Var1,Body2,V,Var2),
      union(Body1,Body2,Body3),
      append([V],TS1,TS2),!.

t_built_in(not_equal(E1,E2),Var1,Body3,not_equal(V1,V2),Var3)
   :- t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),!.
t_built_in(equal(E1,E2),Var1,Body3,equal(V1,V2),Var3)
   :- t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),!.
t_built_in(less_than(E1,E2),Var1,Body3,less_than(V1,V2),Var3)
   :- t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),!.
t_built_in(less_equal(E1,E2),Var1,Body3,less_equal(V1,V2),Var3)
   :- t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),!.
t_built_in(greater_than(E1,E2),Var1,Body3,less_than(V2,V1),Var3)
   :- t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),!.
t_built_in(greater_equal(E1,E2),Var1,Body3,less_equal(V2,V1),Var3)
   :- t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),!.



t_expr(plus(E1,E2),Var,Body4,V,Var3)
   :- Var1 = Var + 1,          % introduction of new variabel.
      str_int(NewName,Var1),
      V = var(NewName),
      t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),
      union([pos(equal(plus(V1,V2),V))],Body3,Body4),!.
t_expr(minus(E1,E2),Var,Body4,V,Var3)
   :- Var1 = Var + 1,          % introduction of new variabel.
      str_int(NewName,Var1),
      V = var(NewName),
      t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),
      union([pos(equal(plus(V,V2),V1))],Body3,Body4),!.
t_expr(mult(E1,E2),Var,Body4,V,Var3)
   :- Var1 = Var + 1,          % introduction of new variabel.
      str_int(NewName,Var1),
      V = var(NewName),
      t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),
      union([pos(equal(mult(V1,V2),V))],Body3,Body4),!.
t_expr(div(E1,E2),Var,Body4,V,Var3)
   :- Var1 = Var + 1,          % introduction of new variabel.
      str_int(NewName,Var1),
      V = var(NewName),
      t_expr(E1,Var1,Body1,V1,Var2),
      t_expr(E2,Var2,Body2,V2,Var3),
      union(Body1,Body2,Body3),
      union([pos(equal(mult(V,V2),V1))],Body3,Body4),!.
t_expr(sign(E1),Var,Body2,V,Var2)
   :- Var1 = Var + 1,          % introduction of new variabel.
      str_int(NewName,Var1),
      V = var(NewName),
      t_expr(E1,Var1,Body1,V1,Var2),
      union([pos(equal(sign(V1),V))],Body1,Body2),!.
t_expr(Expr,Var,[],Expr,Var) :- !.
```

G.7 MODULE qsqr.pro

```
/*
constants
```

```
   localcompilation = 1
*/

ifndef localcompilation

project "datalog"
include "globals.pro"
include "settools.pro"
include "atools.pro"

elsedef

include "globals.pro"
include "settools.pro"
include "mat.pro"
include "write.pro"
include "atools.pro"

enddef

/*
constants
   verbosity = 1
*/

PREDICATES
substitute_term (Expr,Substitution,Expr)
substitute_termlist (Termlist,Substitution,Termlist)
substitute_expr (Expr,Substitution,Expr)
substitute_atom (Atom,Substitution,Atom)
substitute_literal (Literal,Substitution,Literal)
substitute_body (Body,Substitution,Body)

composition (Substitution,Substitution,Substitution)
   composition2 (Substitution,Substitution,Substitution)
   composition3 (Substitution,Substitution,Substitution)

   eliminating1 (Substitution,Substitution,Substitution,Substitution)
   eliminating2 (Substitution,Substitution)
   eliminating3 (SubBinding,Substitution,Substitution)

mgu (Substitution,Termlist,Termlist,Substitution)
   mgu2 (Substitution,Expr,Expr,Termlist,Termlist,Substitution)


filter (Termlist,Forest,Termlist,Forest,Substitutions)
   filter2 (Substitution,Termlist,Termlist,Forest,Termlist,Forest,
    Substitutions)
   form_query (Termlist,Termlist,Termlist)
   form_term (Expr,Expr,Expr)
   rename (Integer,Termlist,Termlist,Termlist)
```

```
binding_info (TSS,BL)
   all_bundet (TSS,BL)
   all_bundet2 (Termlist,BL)
   bindingsliste (TSS,BL,BL)
   bindingstuple (Termlist,BL,BL)

find_sigma (Termlist,Substitutions,Substitutions)
   find_sigma2 (Termlist,Substitutions,Substitutions)
   find_sigma3 (Termlist,Substitution,Substitution)

make_table (Termlist,TSS,Termlist,TSS)
   make_table2 (Termlist,TSS,TSS,TSS)
   make_table3 (Termlist,Termlist,Termlist,Termlist,Termlist)

var_bound (Expr,Termlist,BL,Integer)
arg_bound (Termlist,Termlist,BL,Integer)

all_ready (Body,Termlist,BL,Body,Body)
ready (Literal,Termlist,BL)
ready_neg (Atom,Termlist,BL)
ready_pos (Atom,Termlist,BL)
default_adornment (Termlist,Adornlist)
ready_termlist (Termlist,Termlist,BL,Adornlist)
ready_term (Expr,Termlist,BL,AdornBinding)
ready_cases (Integer,AdornBinding)

beta (Literal,Termlist,BL,Real)
beta_test (INTEGER,INTEGER,INTEGER,REAL)
count_facts (String,Integer)

selection_function (Termlist,BL,Body,Literal,Body)
sol (Body,Termlist,BL,Literal,Body)
sol_test (Literal,Literal,Termlist,BL,Literal,Literal)
sol_comp (Literal,Real,Literal,Real,Literal,Literal)

  vis_beta (Body,Termlist,BL)


  pre_qsqr (Termlist,Termlist)
  convert_qtermlist (QTermlist,Termlist)
  convert_qexpr (QExpr,Expr)



  qsqr (relation,relation)



consult_edb (relation,relation)
match1 (forest,forest,forest)
```

```
match2 (Termlist,forest,forest,forest)
match3 (Termlist,Termlist)

non_recursive_group (relation,Dbase)
recursive_group (relation,Dbase)
only_id (STRING,Dbase,Dbase)

within_range (relation,relation)
within_range_tuples (STRING,Forest,Forest)
within_range_tuple (INTEGER,STRING,Termlist)
within_range_term (Expr,Number,Number)

store_lemma (relation)
store_lemma_forest (STRING,forest)
store_lemma_tree (STRING,Termlist)


consult_recursive (relation,Dbase,relation)
consult_recursive2 (relation,relation,Dbase,relation)
consult_recursive3 (relation,Dbase,relation)
consult_idb (relation,Dbase,relation)
consult_rule (relation,Rule,relation)

pre_loop (Termlist,Forest,Body,Termlist,Forest)
  av_tree (Termlist,tree)
  minus_already_known (Termlist,Termlist,Termlist)
all_var_body (Body,Termlist)
all_var_literal (Literal,Termlist)
all_var_atom (Atom,Termlist)
all_var_termlist (Termlist,Termlist)
all_var_expr (Expr,Termlist)


loop (relation,Body,relation)

cases_literal (Literal,relation,relation)



pre_call (relation,Atom,relation)
projectt_tuples (Termlist,Forest,Termlist,Forest)
projectt_tuple (Termlist,Termlist,Termlist,Termlist)

select_rel (relation,Atom,relation) % (i,i,o)
select_tuples (Termlist,forest,Atom,forest) % (i,i,i,o)
tss2forest (TSS,forest)
forest2tss (forest,TSS)

select_tuple (Termlist,Termlist,Atom,TSS)
cases_values_plus (Expr,Expr,Expr,Expr,Expr,Expr,Termlist,Termlist,TSS)
cases_values_mult (Expr,Expr,Expr,Expr,Expr,Expr,Termlist,Termlist,TSS)
```

```
cases_values_sign (Expr,Expr,Expr,Expr,Termlist,Termlist,TSS)
compare_values (STRING,Expr,Expr,Termlist,TSS)


read_value (Expr,Termlist,Termlist,Expr)
write_value (Expr,Expr,Expr,Termlist,Termlist,TSS)

vis_relation (relation)
vis_forest (forest)


CLAUSES


substitute_term(T,Phi,Ti) % ceri p83
   :- member(binding(T,Ti),Phi), not(Ti = anonymousvariable()), !.
    % FP T,Ti eller Ti,T
substitute_term(T,_,T).

substitute_termlist([],_,[]) :- !.
substitute_termlist([T1|TS1],Phi,[T2|TS2])
   :- substitute_term(T1,Phi,T2),
      substitute_termlist(TS1,Phi,TS2).

substitute_expr(plus(T1,T2),Phi,plus(S1,S2))
   :- substitute_term(T1,Phi,S1),
      substitute_term(T2,Phi,S2), !.
substitute_expr(mult(T1,T2),Phi,mult(S1,S2))
   :- substitute_term(T1,Phi,S1),
      substitute_term(T2,Phi,S2), !.
substitute_expr(sign(T1),Phi,sign(S1))
   :- substitute_term(T1,Phi,S1), !.

substitute_atom(head(ID,TS1),Phi,head(ID,TS2))
   :- substitute_termlist(TS1,Phi,TS2), !.
substitute_atom(less_than(T1,T2),Phi,less_than(S1,S2))
   :- substitute_term(T1,Phi,S1),
      substitute_term(T2,Phi,S2), !.
substitute_atom(less_equal(T1,T2),Phi,less_equal(S1,S2))
   :- substitute_term(T1,Phi,S1),
      substitute_term(T2,Phi,S2), !.
substitute_atom(not_equal(T1,T2),Phi,not_equal(S1,S2))
   :- substitute_term(T1,Phi,S1),
      substitute_term(T2,Phi,S2), !.
substitute_atom(equal(T1,T2),Phi,equal(S1,S2))
   :- substitute_expr(T1,Phi,S1),
      substitute_term(T2,Phi,S2), !.

substitute_literal(neg(Atom1),Phi,neg(Atom2))
   :- substitute_atom(Atom1,Phi,Atom2), !.
substitute_literal(pos(Atom1),Phi,pos(Atom2))
```

```
   :- substitute_atom(Atom1,Phi,Atom2), !.

substitute_body([Literal1|Body1],Phi,Body3)
   :- substitute_literal(Literal1,Phi,Literal2),
      substitute_body(Body1,Phi,Body2),
      insert(Literal2,Body2,Body3), !.
substitute_body([],_,[]).



composition(Phi,Sigma,Result)
   :-composition2(Phi,Sigma,Result1),
     eliminating1(Phi,Sigma,Result1,Result2),
     eliminating2(Result2,Result).


   composition3(Phi,Sigma,Result)
      :- append(Phi,Sigma,Result).

   composition2([],Sigma,Sigma) :- !.
   composition2([binding(Var,Term1)|Tail],Sigma,
         [binding(Var,Term2)|Result])
      :- substitute_term(Term1,Sigma,Term2),
         composition2(Tail,Sigma,Result).

   eliminating1(_,[],Result,Result) :- !.
   eliminating1(Phi,[binding(E1,E2)|Sigma],Result1,Result3)
      :- member(binding(E1,_),Phi), !,
         eliminating3(binding(E1,E2),Result1,Result2),
         eliminating1(Phi,Sigma,Result2,Result3), !.
   eliminating1(Phi,[_|Sigma],Result1,Result2)
      :- eliminating1(Phi,Sigma,Result1,Result2), !.

   eliminating3 (_,[],[]) :- !.
   eliminating3 (T,[T|R],S) :- eliminating3(T,R,S),!.
   eliminating3 (T,[A|R],[A|S]) :- eliminating3(T,R,S).

   eliminating2 ([],[]) :- !.
   eliminating2 ([binding(E,E)|Rest1],Rest2)
    :- eliminating2(Rest1,Rest2), !.
   eliminating2 ([B|Rest1],[B|Rest2])
    :- eliminating2(Rest1,Rest2), !.



mgu(Phi,[],[],Phi) :- /*vis_resultat2(Phi),*/ !.
mgu(Phi,[T1|TS1],[T2|TS2],Sigma)
   :- substitute_term(T1,Phi,S1),
      substitute_term(T2,Phi,S2),
      mgu2(Phi,S1,S2,TS1,TS2,Sigma).
   mgu2(Phi,T,T,TS1,TS2,Sigma)
      :- mgu(Phi,TS1,TS2,Sigma), !.
   mgu2(Phi,T1,T2,TS1,TS2,Sigma)
      :- T2 = var(_), !,
         composition(Phi,[binding(T2,T1)],Phi1),
         mgu(Phi1,TS1,TS2,Sigma), !.
   mgu2(Phi,_,T2,TS1,TS2,Sigma)
      :- T2 = anonymousvariable(), !,
         mgu(Phi,TS1,TS2,Sigma), !.

   mgu2(Phi,T1,T2,TS1,TS2,Sigma)
      :- T1 = var(_), !,
         composition(Phi,[binding(T1,T2)],Phi1),
         mgu(Phi1,TS1,TS2,Sigma), !.
   mgu2(Phi,T1,_,TS1,TS2,Sigma)
      :- T1 = anonymousvariable(), !,
         mgu(Phi,TS1,TS2,Sigma), !.
   mgu2(_,_,_,_,_,[dummy])
      :-
ifdef verbosity
      vis_resultat2([dummy]),
enddef
      !.

filter(_,[],_,[],[]) :- !.
filter(QTS,QForest1,RTS,RForest,Subst2)
   :- pick_tree(QForest1,QTree,QForest2),
      termlist_tree(TS,QTree),
      form_query(QTS,TS,GQTS),
      mgu([],RTS,GQTS,Phi),
      filter2(Phi,QTS,GQTS,QForest2,RTS,RForest,Subst1),
      insert(Phi,Subst1,Subst2), !.
filter(QTS,QForest,RTS,[],[])
   :- nl,write("fail filter: ",QTS,QForest,RTS), fail.

   filter2([dummy],QTS,_,QForest,RTS,RForest,Subst)
      :- filter(QTS,QForest,RTS,RForest,Subst), !.
   filter2(Phi,QTS,GQTS,QForest,RTS,RForest2,Subst)
      :- substitute_termlist(GQTS,Phi,GRTS),
         pre_qsqr(GRTS,TS),
         termlist_tree(TS,RTree),
         filter(QTS,QForest,RTS,RForest1,Subst),
         insert_tree(RTree,RForest1,RForest2).


   form_query([],[],[]) :- !.
   form_query([T1|TS1],[T2|TS2],[T3|TS3])
      :- form_term(T1,T2,T3),
         form_query(TS1,TS2,TS3).

   form_term(Q,anonymousvariable,Q) :- !.
   form_term(_,CON,CON).


rename(_,[],_,[]) :- !.
rename(Var,[T1|TS1],TS2,[var(VarS)|Tail])
   :- member(T1,TS2), !,
      Var + 1 = Var1,
      str_int(VarS,Var1),
      substitute_termlist(TS1,[binding(T1,var(VarS))],TS3),
      rename(Var1,TS3,TS2,Tail),!.
rename(Var,[Head|TS1],TS2,[Head|Tail])
   :- rename(Var,TS1,TS2,Tail),!.



binding_info(TSS,BL)
   :- all_bundet(TSS,Fri),
      bindingsliste(TSS,Fri,BL),!.
binding_info(TSS,[])
   :- nl, write("fail binding_info: "),
      nl, write(indent,"TSS: ",TSS), fail.


all_bundet([TS|_],BL)
   :- all_bundet2(TS,BL).

all_bundet2([_|TS],[bundet|BL])
   :- all_bundet2(TS,BL),!.
all_bundet2([],[]).

bindingsliste([TS|TSS],BL1,BL3)
   :- bindingstuple(TS,BL1,BL2),
      bindingsliste(TSS,BL2,BL3), !.
bindingsliste([],BL,BL).

bindingstuple([str(_)|TS],[bundet|BL1],[bundet|BL2])
   :- bindingstuple(TS,BL1,BL2), !.
bindingstuple([q(_,_,_)|TS],[bundet|BL1],[bundet|BL2])
   :- bindingstuple(TS,BL1,BL2), !.
bindingstuple([_|TS],[_|BL1],[fri|BL2])
   :- bindingstuple(TS,BL1,BL2), !.
bindingstuple([],[],[]).

find_sigma(PTermlist,Subst,Sigma)
   :- find_sigma2(PTermlist,Subst,Sigma).

   find_sigma2(PTermlist,[Phi1|Sigma1],Sigma3)
      :- find_sigma3(PTermlist,Phi1,Phi2),
         find_sigma2(PTermlist,Sigma1,Sigma2),!,
         insert(Phi2,Sigma2,Sigma3), !.
   find_sigma2(_,[],[]) :- !.
   find_sigma3(PTermlist,[binding(var(V),var(W))|Phi1],
    [binding(var(V),var(W))|Phi2])
      :- member(var(V),PTermlist), !,
         find_sigma3(PTermlist,Phi1,Phi2), !.
   find_sigma3(PTermlist,[_|Phi1],Phi2)
      :- find_sigma3(PTermlist,Phi1,Phi2), !.
   find_sigma3(_,[],[]) :- !.


/* domains
make_table (Termlist,TSS,Termlist,TSS)
   make_table2 (Termlist,TSS,TSS,TSS)
   make_table3 (Termlist,Termlist,Termlist,Termlist,Termlist)
*/

make_table(Termlist,ListSet,Head,Base)
   :- make_table2(Termlist,ListSet,HeadSet,Base),
      member(Head,HeadSet),!.
make_table(Termlist,TSS,Termlist,TSS)
   :- nl, write("fail make_table: "),
      nl, write(indent,"Termlist: ",Termlist),
      nl, write(indent,"TSS:      ",TSS).
      %fail.

make_table2(TS1,[TS2|TSS],TSS3,TSS4)
   :- make_table3(TS1,TS2,[],TS3,TS4),
      make_table2(TS1,TSS,TSS1,TSS2),!,
      insert(TS3,TSS1,TSS3),
      insert(TS4,TSS2,TSS4), !.
make_table2(_,[],[],[]).

make_table3([E1|List1],[_|List2],Set,List3,List4)
   :- member(E1,Set), !,
      make_table3(List1,List2,Set,List3,List4),!.
make_table3([E1|List1],[E2|List2],Set1,[E1|List3],[E2|List4])
   :- insert(E1,Set1,Set2),
      make_table3(List1,List2,Set2,List3,List4),!.
make_table3([],[],_,[],[]).



var_bound(_,[],_,0) :- !.
var_bound(VAR,[VAR|_],[bundet|_],1) :- !.
var_bound(VAR,[VAR|_],[fri|_],0) :- !.
var_bound(VAR,[_|TS],[_|Binding],I) :- var_bound(VAR,TS,Binding,I), !.

arg_bound([],_,_,0) :- !.
arg_bound([T1|TS1],TS2,Binding,I)
   :- T1 = var(_),
      arg_bound(TS1,TS2,Binding,A),
      var_bound(T1,TS2,Binding,B),
      I = A + B, !.
arg_bound([T1|TS1],TS2,Binding,A)
   :- T1 = anonymousvariable(),
      arg_bound(TS1,TS2,Binding,A),!.
arg_bound([_|TS1],TS2,Binding,A1)
   :- arg_bound(TS1,TS2,Binding,A),
      A + 1 = A1, !.

all_ready([Literal|Body1],Termlist,Binding,[Literal|Ready],NotReady)
   :- ready(Literal,Termlist,Binding), !,
      all_ready(Body1,Termlist,Binding,Ready,NotReady), !.
all_ready([Literal|Body1],Termlist,Binding,Ready,[Literal|NotReady])
   :- all_ready(Body1,Termlist,Binding,Ready,NotReady), !.
all_ready([],_,_,[],[]).


ready(neg(Head),Termlist,Binding)
   :- ready_neg(Head,Termlist,Binding), !.
ready(pos(Atom),Termlist,Binding)
   :- ready_pos(Atom,Termlist,Binding), !.


ready_neg(head(_,TS1),TS2,Binding)
   :- arg_bound(TS1,TS2,Binding,B),
      card(TS1,C),!, C=B.

ready_pos(head(ID,TS1),TS2,Binding)
   :- adornment_db(ID,BF),
      ready_termlist(TS1,TS2,Binding,BF),!.
ready_pos(head(_,[]),[],_) :- !.
ready_pos(head(_,TS1),TS2,Binding)
   :- default_adornment(TS1,BF),
      ready_termlist(TS1,TS2,Binding,BF),!.

ready_pos(less_than(T1,T2),TS,Binding)
   :- arg_bound([T1,T2],TS,Binding,B), !, B=2, !.
ready_pos(less_equal(T1,T2),TS,Binding)
   :- arg_bound([T1,T2],TS,Binding,B), !, B=2, !.
ready_pos(not_equal(T1,T2),TS,Binding)
   :- arg_bound([T1,T2],TS,Binding,B), !, B=2, !.
ready_pos(equal(plus(T1,T2),T3),TS,Binding)
   :- arg_bound([T1,T2,T3],TS,Binding,B), !, B >= 2, !.
ready_pos(equal(mult(T1,T2),T3),TS,Binding)
   :- arg_bound([T1,T2,T3],TS,Binding,B), !, B >= 2, !.
ready_pos(equal(sign(T1),T2),TS,Binding)
   :- arg_bound([T1,T2],TS,Binding,B), !, B >= 1, !.


default_adornment([_|TS],[fb()|BF])
   :- default_adornment(TS,BF), !.
```

```
default_adornment([],[]).


ready_termlist([T1|TS1],TS2,Binding,[B|BF])
   :- ready_term(T1,TS2,Binding,B),
      ready_termlist(TS1,TS2,Binding,BF).
ready_termlist([],_,_,_).


ready_term(anonymousvariable,_,_,_) :- !.
ready_term(str(_),_,_,_) :- !.
ready_term(q(_,_,_),_,_,_) :- !.
ready_term(T1,TS2,Binding,B)
   :- var_bound(T1,TS2,Binding,I),
      ready_cases(I,B).

ready_cases(0,fb()) :- !.
ready_cases(0,ff()) :- !.
ready_cases(1,_) :- !.


beta(neg(_),_,_,1) :- !.
beta(pos(head(_,[])),_,_,1) :- !.
beta(pos(head(ID,TS)),Termlist,Binding,Beta)
   :- card(TS,A),
      arg_bound(TS,Termlist,Binding,B),
      count_facts(ID,C),
      beta_test(A,B,C,Beta),
ifdef verbosity
      nl,write("BETA: ",ID,"  beta = ",
      B,"/(",A,"*",C,") = ",Beta),
enddef
      !.
beta(pos(_),_,_,1) :- !.

beta_test(_,_,0,0) :- !.
beta_test(A,B,C,Beta) :- Beta = B / (A*C), !.


count_facts(ID,C1)
   :- fact_edb(ID,TS), !,
      retract(fact_edb(ID,TS)),
      count_facts(ID,C),
      assert(fact_edb(ID,TS)),
      C1 = C + 1, !.
count_facts(_,0).



ifdef verbosity
selection_function(Termlist,Binding,Body1,Literal,Body)
   :- Body1 = [Literal|Body],
      nl,write("call selection_function: "),
      nl,write(indent,"Body:     ",Body1),
      nl,write(indent,"Termlist: ",Termlist),
      nl,write(indent,"Binding:  ",Binding),
      fail.
enddef

selection_function(_,_,[Literal],Literal,[]) :- !.
selection_function(Termlist,Binding,Body,SelectedLiteral,ReturnBody)
   :- all_ready(Body,Termlist,Binding,ReadyBody,NotReadyBody),
      sol(ReadyBody,Termlist,Binding,SelectedLiteral,NotSelectedLiterals),
      union(NotSelectedLiterals,NotReadyBody,ReturnBody),
ifdef verbosity
      nl,write("return selection_function: "),
      nl,write(indent,"ReturnBody: ",ReturnBody),
      nl,write(indent,"Selected Literal: ",SelectedLiteral),
enddef
      !.

selection_function(Termlist,Binding,Body1,Literal,Body)
   :- Body1 = [Literal|Body],
      nl,write("fail selection_function: "),
      nl,write(indent,"Body:     ",Body1),
      nl,write(indent,"Termlist: ",Termlist),
      nl,write(indent,"Binding:  ",Binding),
      fail.



/*  sol - Select One Literal   */
sol([Literal],_,_,Literal,[]) :- !.
sol([Literal1|Body1],TS,BD,Winner,Body3)
   :- sol(Body1,TS,BD,Literal2,Body2),
      sol_test(Literal1,Literal2,TS,BD,Winner,Loser),
      insert(Loser,Body2,Body3).

sol_test(L1,L2,TS,BD,Winner,Loser)
   :- beta(L1,TS,BD,Beta1),
      beta(L2,TS,BD,Beta2),
      sol_comp(L1,Beta1,L2,Beta2,Winner,Loser).

sol_comp(L1,Beta1,L2,Beta2,L1,L2) :- Beta1 > Beta2, !.
sol_comp(L1,_,L2,_,L2,L1).


vis_beta([],_,_) :- nl,!.
vis_beta([H|R],TS,B)
   :- vis_literal(H),
      beta(H,TS,B,Beta),
      write(indent,"Beta: ",Beta),nl,
      vis_beta(R,TS,B).
```

```
interpret_program(query(pos(head(QID,QTS))),Rel2)
   :- convert_qtermlist(QTS,RTS),

      pre_qsqr(RTS,TS), termlist_tree(TS,Tree), Forest = [Tree],

      Rel1 = r(QID,RTS,Forest),
      qsqr(Rel1,Rel2),!.
interpret_program(query(pos(head(QID,QTS))),r(QID,TS,[]))
   :- convert_qtermlist(QTS,RTS),
      pre_qsqr(RTS,TS).


convert_qtermlist([QExpr|QTermlist],[Expr|Termlist])
   :- convert_qexpr(QExpr,Expr),
      convert_qtermlist(QTermlist,Termlist),!.
convert_qtermlist([],[]).

convert_qexpr(var(S),var(S)).
convert_qexpr(signrat(I1,I2),q(-1,I1,I2)).
convert_qexpr(rat(I1,I2),q(1,I1,I2)).
convert_qexpr(sign(I),q(-1,I,1)).
convert_qexpr(int(I),q(1,I,1)).
convert_qexpr(anonymousvariable(),anonymousvariable()).
convert_qexpr(str(S),str(S)).


pre_qsqr([var(_)|R1],[anonymousvariable|R2]) :- pre_qsqr(R1,R2),!.
pre_qsqr([Term|R1],[Term|R2]) :- pre_qsqr(R1,R2),!.
pre_qsqr([],[]).

/*FINN*/
qsqr(r(ID,Termlist,_),r(ID,Termlist,[]))
   :- rulegraph_db(init(ID)),!.
qsqr(r(ID,Termlist,_),r(ID,Termlist,[]))
   :- %storage(StackSize,HeapSize,TrailSize),
 storage(StackSize,_       ,_       ),
StackSize < 2500,
nl, storage,
nl, write("Warning: Recursion
          terminated because of LOW STACK"),!.
qsqr(Rel,Rel9)
   :- within_range(Rel,Rel1),
      consult_edb(Rel1,Rel2),
      within_range(Rel2,Rel3),
ifdef verbosity
      write("edb: "), vis_relation(Rel2),nl,
enddef
      non_recursive_group(Rel1,Group1),
ifdef verbosity
```

```
      write("non recursive group: "), nl, vis_db(Group1),nl,
enddef
      consult_idb(Rel1,Group1,Rel4),
      within_range(Rel4,Rel5),
      store_lemma(Rel5),
ifdef verbosity
      nl,write("after consult idb: "), vis_relation(Rel5),nl,
enddef
      relation_union(Rel3,Rel5,Rel8),

      recursive_group(Rel1,Group2),
ifdef verbosity
      write("recursive group: "), nl, vis_db(Group2),nl,
enddef
      consult_recursive(Rel1,Group2,Rel6),
      within_range(Rel6,Rel7),
ifdef verbosity
      nl,write("after consult recursive: "), vis_relation(Rel7),nl,
enddef
      relation_union(Rel8,Rel7,Rel9),
ifdef verbosity
      write("SUCCES ANSWER TO QSQR: "), vis_relation(Rel9),
enddef
      !.
qsqr(Rel,Rel)
   :- nl, write("QSQR failed: "), vis_relation(Rel), nl, fail.


consult_edb(r(ID,[],_),r(ID,[],[leaf]))
   :- fact_edb(ID,[]),!.
consult_edb(r(ID,TS,Forest),r(ID,TS,Facts3))
   :- findall(Termlist,fact_edb(ID,Termlist),TSS),
      tss2forest(TSS,Facts1),
      match1(Forest,Facts1,Facts2),
      filter(TS,Facts2,TS,Facts3,_),!.
       % Se specification p80 app D.1.4
consult_edb(r(ID,TS,_),r(ID,TS,[])).


match1([],_,[]) :- !.
match1(_,[],[]) :- !.
match1(Forest,Facts,Forest3)
   :- pick_tree(Forest,Tree,Forest1),
      termlist_tree(Termlist,Tree),
      match2(Termlist,Facts,F1,F2),
      match1(Forest1,F2,Forest2),
      forest_union(F1,Forest2,Forest3).

match2(_,[],[],[]) :- !.
match2(Termlist,Facts,F3,F2)
   :- pick_tree(Facts,Fact,Facts1),
      termlist_tree(Fact1,Fact),
      match3(Termlist,Fact1),
      match2(Termlist,Facts1,F1,F2),
      insert_tree(Fact,F1,F3), !.
match2(Termlist,Facts,F1,F3)
   :- pick_tree(Facts,Fact,Facts1),
      match2(Termlist,Facts1,F1,F2),
      insert_tree(Fact,F2,F3), !.

match3([],[]) :- !.
match3([anonymousvariable|QTS],[_|Fact])
   :- match3(QTS,Fact),!.
match3([Match|QTS],[Match|Fact])
   :- match3(QTS,Fact),!.
match3([var(_)|QTS],[_|Fact])
   :- match3(QTS,Fact).






non_recursive_group(r(ID,_,_),Group)
   :- rulegraph_db(step(Group)),
      member(rule(head(ID,_),_),Group),!.
non_recursive_group(_,[]).

recursive_group(r(ID,_,_),Group)
   :- rulegraph_db(repeat(Group1)),
      member(rule(head(ID,_),_),Group1),!,
      only_id(ID,Group1,Group),!.
recursive_group(_,[]).


only_id(_,[],[]) :- !.
only_id(ID,[Rule|Rules1],[Rule|Rules2])
   :- Rule = rule(head(ID,_),_),
      only_id(ID,Rules1,Rules2),!.
only_id(ID,[_|Rules1],Rules2)
   :- only_id(ID,Rules1,Rules2),!.



within_range(r(ID,TS,Forest),r(ID,TS,Forest))
   :- not(range_db(ID,_,_,_)),!.
within_range(r(ID,TS,Forest1),r(ID,TS,Forest2))
   :- within_range_tuples(ID,Forest1,Forest2).

within_range_tuples(_,[],[]) :- !.
within_range_tuples(ID,Forest1,Forest4)
   :- pick_tree(Forest1,Tree,Forest2),
      termlist_tree(Termlist,Tree),
      within_range_tuple(1,ID,Termlist),
      within_range_tuples(ID,Forest2,Forest3),
      insert_tree(Tree,Forest3,Forest4),!.
within_range_tuples(ID,Forest1,Forest3)
   :- pick_tree(Forest1,_,Forest2),
      within_range_tuples(ID,Forest2,Forest3).



within_range_tuple(_,_,[]) :- !.
within_range_tuple(POS,ID,[_|TS])
   :- not(range_db(ID,POS,_,_)),
      Next = POS + 1,
      within_range_tuple(Next,ID,TS),!.
within_range_tuple(POS,ID,[T|TS])
   :- range_db(ID,POS,Low,Hi),!,
      within_range_term(T,Low,Hi),
      Next = POS + 1,
      within_range_tuple(Next,ID,TS).

within_range_term(anonymousvariable(),_,_) :- !.
within_range_term(str(_),_,_) :- !.
within_range_term(Q,Low,Hi)
   :- Q = q(_,_,_),
      rational2real(Q,R),
      number2real(Low,LowR),
      number2real(Hi,HiR),
      LowR <= R, R <= HiR,!.
ifdef verbosity
within_range_term(Term,Low,Hi)
   :- nl, write("Term outside range: ", Term),
      nl, write(indent,"Low: ",Low,"  Hi: ",Hi), fail.
enddef
store_lemma(r(ID,_,Forest))
   :- store_lemma_forest(ID,Forest).

store_lemma_forest(_,[]) :- !.
store_lemma_forest(ID,Forest1)
   :- pick_tree(Forest1,Tree,Forest2),
      termlist_tree(TS,Tree),
      store_lemma_tree(ID,TS),
      store_lemma_forest(ID,Forest2).

store_lemma_tree(ID,TS)
   :- fact_edb(ID,TS), !.
store_lemma_tree(ID,TS)
   :- assert(fact_edb(ID,TS)).


consult_recursive(r(ID,TS,_),[],r(ID,TS,[])) :- !.
consult_recursive(r(ID,TS,Forest),_,r(ID,TS,[]))
   :-  query_db(ID,Forest),!,
ifdef verbosity
       nl,write("NON-ADMISSIBLE QUERY: ", ID,TS),
       nl,vis_forest(Forest),nl,
enddef
       !.

consult_recursive(Rel1,Group,Rel2)
   :- Rel1 = r(ID,_,Forest),
      %retractall(query_db(ID,_),_),
      asserta(query_db(ID,Forest)),
      consult_idb(Rel1,Group,Rel2),
      store_lemma(Rel2).
     % consult_recursive2(Rel1,Rel2,Group,Rel3)./*FINN*/

ifdef verbosity
consult_recursive2(Rel1,Rel2,Groupe,Rel2)
   :- nl, write("CALL recursive2: "),
      nl, write(indent,"Rel1: "), vis_relation(Rel1),
      nl, write(indent,"Rel2: "), vis_relation(Rel2),
      nl, write(indent,"Groupe: ",Groupe),
      fail.

consult_recursive2(Rel,Rel,_,Rel)
   :- nl,write("RECURSION STOPED BY consult_recursive2."),!.
enddef
consult_recursive2(_,Rel2,Group,Rel3)
   :- consult_recursive3(Rel2,Group,Rel3).

consult_recursive3(Rel1,Group,Rel3)
   :- consult_idb(Rel1,Group,Rel2),
      store_lemma(Rel2),
      consult_recursive2(Rel1,Rel2,Group,Rel3).




/* old version

consult_recursive(Rel1,Group,Rel3)
   :- consult_idb(Rel1,Group,Rel2),
      consult_recursive2(Rel1,Rel2,Group,Rel3).

consult_recursive2(Rel,Rel,_,Rel) :- !.
consult_recursive2(_,Rel2,Group,Rel3)
   :- consult_recursive(Rel2,Group,Rel3).
*/
```

```
/*FINN*/
consult_idb(r(ID,TS,_),[],r(ID,TS,[])) :- !.
consult_idb(Rel,[Rule|Rules],Rel3)
   :- consult_rule(Rel,Rule,Rel1),
      consult_idb(Rel,Rules,Rel2),
      relation_union(Rel1,Rel2,Rel3),!.
consult_idb(Rel,Group,Rel)
   :- nl, write("fail consult_idb: "),
      nl, write(indent,"Rel: "), vis_relation(Rel),
      nl, write(indent,"Group: ",Group), fail.


/*FINN*/
consult_rule(r(ID,TS,[]),_,r(ID,TS,[])) :- !.
consult_rule(r(ID,QTS1,QForest),rule(head(ID,RTS),Body),Answer)
   :-  rename (999,QTS1,RTS,QTS2),
        filter (QTS2,QForest,RTS,RForest,Subst),
find_sigma (RTS,Subst,Sigmas),
member (Phi,Sigmas), !,
substitute_termlist(RTS,Phi,RTS1),
substitute_body (Body,Phi,Body1),

pre_loop(RTS1,RForest,Body1,RTS2,RForest2),

loop(r(ID,RTS2,RForest2),Body1,Rel),

Rel = r(P,TS,TSS),
projectt_tuples(TS,TSS,RTS,TSSOut),

Answer = r(P,QTS1,TSSOut),

!.
consult_rule(Rel,Rule,Rel)
   :- nl, write("fail consult_rule: "),
      nl, write("Rel: "), vis_relation(Rel),
      nl, write("Rule: ",Rule),fail.


pre_loop(RTS1,RForest1,Body1,RTS2,RForest2)
   :- all_var_body(Body1,V1),
      minus_already_known(V1,RTS1,V2),
      av_tree(V2,AVTree),
      append(V2,RTS1,RTS2),
      cartecian_product([AVTree],RForest1,RForest2).


  av_tree ([_|Rest],t(anonymousvariable(),[T]))
     :- av_tree(Rest,T), !.
  av_tree ([],leaf).

  minus_already_known([],_,[]) :- !.
  minus_already_known([E|V1],RTS,V2)
     :- member(E,RTS),
        minus_already_known(V1,RTS,V2), !.
  minus_already_known([E|V1],RTS,[E|V2])
     :- minus_already_known(V1,RTS,V2), !.



all_var_body([],[]) :- !.
all_var_body([L|Body],V)
   :- all_var_literal(L,V1),
      all_var_body(Body,V2),
      union(V1,V2,V).

all_var_literal(pos(Atom),V)
   :- all_var_atom(Atom,V), !.
all_var_literal(neg(Atom),V)
   :- all_var_atom(Atom,V).


all_var_atom(head(_,TS),V)
   :- all_var_termlist(TS,V), !.
all_var_atom(less_than(E1,E2),V)
   :- all_var_expr(E1,V1),
      all_var_expr(E2,V2),
      union(V1,V2,V), !.
all_var_atom(less_equal(E1,E2),V)
   :- all_var_expr(E1,V1),
      all_var_expr(E2,V2),
      union(V1,V2,V), !.
all_var_atom(not_equal(E1,E2),V)
   :- all_var_expr(E1,V1),
      all_var_expr(E2,V2),
      union(V1,V2,V), !.
all_var_atom(equal(E1,E2),V)
   :- all_var_expr(E1,V1),
      all_var_expr(E2,V2),
      union(V1,V2,V), !.

all_var_termlist([],[]) :- !.
all_var_termlist([E|TS],V)
   :- all_var_termlist(TS,V1),
      all_var_expr(E,V2),
      union(V1,V2,V).


all_var_expr(plus(E1,E2),V)
   :- all_var_expr(E1,V1),
      all_var_expr(E2,V2),
      union(V1,V2,V), !.
all_var_expr(mult(E1,E2),V)
```

```
   :- all_var_expr(E1,V1),
      all_var_expr(E2,V2),
      union(V1,V2,V), !.
all_var_expr(sign(E),V)
   :- all_var_expr(E,V), !.
all_var_expr(var(V),[var(V)]) :- !.
all_var_expr(_,[]).


ifdef verbosity
loop(Rel,Body,Rel)
   :- nl, write("call loop: "),
      nl, write(indent,"Rel: "), vis_relation(Rel),
      nl, write(indent,"Body: ",Body),
      fail.
enddef
loop(Rel1,[],Rel1) :- !.
loop(r(ID,TS,[]),_,r(ID,TS,[])) :- !.
loop(Rel1,Body1,Rel3)
   :- Rel1 = r(_,RTS,Forest),
      forest2tss (Forest,RTSS),
      make_table (RTS,RTSS,TableHead,TableBase),
      binding_info (TableBase,BL),
      selection_function(TableHead,BL,Body1,Literal,Body),
      cases_literal(Literal,Rel1,Rel2),
      loop(Rel2,Body,Rel3),!.
loop(Rel,Body,Rel)
   :- Rel = r(_,RTS,Forest),
      forest2tss (Forest,RTSS),
      make_table (RTS,RTSS,TableHead,TableBase),
      binding_info (TableBase,BL),
      nl, write("fail loop first: "),
      nl, write(indent,"Rel: "), vis_relation(Rel),
      nl, write(indent,"Body: ",Body),
      nl, write(indent,"RTS: ",RTS),
      nl, write(indent,"Forest: "), vis_forest(Forest),
      nl, write(indent,"RTSS: ",RTSS),
      nl, write(indent,"TableHead: ",TableHead),
      nl, write(indent,"TableBase: ",TableBase),
      nl, write(indent,"Binding: ",BL),
      fail.
loop(Rel,Body,Rel)
   :- nl, write("fail loop second: "),
      nl, write(indent,"Rel: "), vis_relation(Rel),
      nl, write(indent,"Body: ",Body),
      fail.


ifdef verbosity
cases_literal(Literal,Rel,Rel)
   :- nl, write("call cases_literal: "),
      nl, write(indent,"Literal: ",Literal),
      nl, write(indent,"Rel: "), vis_relation(Rel), fail.
enddef

cases_literal(pos(Atom),Rel1,Rel4)
   :- Atom = head(_,_),
      pre_call(Rel1,Atom,Rel2),
      qsqr(Rel2,Rel3),
ifdef verbosity
      nl,write("Before JOIN: "),
      nl,write(indent,"Rel1: "), vis_relation(Rel1),
      nl,write(indent,"Rel2: "), vis_relation(Rel3),
enddef
      join(Rel1,Rel3,Rel4),
ifdef verbosity
      nl,write("After JOIN: "),
      nl,write(indent,"Rel3: "), vis_relation(Rel4),
enddef
      !.
cases_literal(pos(Atom),Rel1,Rel2)
   :- select_rel(Rel1,Atom,Rel2),!.
cases_literal(neg(Atom),Rel1,Rel4)
   :- Atom = head(_,_),
      pre_call(Rel1,Atom,Rel2),
      qsqr(Rel2,Rel3),
ifdef verbosity
      nl,write("Before DIFFERENCE: "),
      nl,write(indent,"Rel1: "), vis_relation(Rel1),
      nl,write(indent,"Rel2: "), vis_relation(Rel3),
enddef
      diff(Rel1,Rel3,Rel4),
ifdef verbosity
      nl,write("After DIFFERENCE: "),
      nl,write(indent,"Rel3: "), vis_relation(Rel4),
enddef
      !.
cases_literal(Literal,Rel,Rel)
   :- nl, write("fail cases_literal: "),
      nl, write(indent,"Literal: ",Literal),
      nl, write(indent,"Rel: "), vis_relation(Rel), fail.


pre_call(r(_,RTS,RTSS),head(ID,TS),r(ID,TS,TSS))
   :- projectt_tuples(RTS,RTSS,TS,TSS).

projectt_tuples(_,[],_,[]) :- !.
projectt_tuples(RTS,Forest1,TS,Forest5)
   :- pick_tree(Forest1,Tree,Forest2),
      Termlist_tree(Tuple,Tree),
      projectt_tuple(RTS,Tuple,TS,TS1),
      pre_qsqr(TS1,TS2),
      Termlist_tree(TS2,Tree2),
      projectt_tuples(RTS,Forest2,TS,Forest4),
      forest_union([Tree2],Forest4,Forest5).


projectt_tuple(_,_,[],[]) :- !.
projectt_tuple(RTS,Tuple,[T|TS1],[Value|TS2])
   :- projectt_tuple(RTS,Tuple,TS1,TS2),
      read_value(T,RTS,Tuple,Value).




select_rel(r(ID,TS,TPS1),Atom,r(ID,TS,TPS2))
   :- select_tuples(TS,TPS1,Atom,TPS2).

select_tuples(_,[],_,[]) :- !.
select_tuples(TS,Forest1,Atom,Forest5)
   :- pick_tree (Forest1,Tree,Forest2),
      Termlist_tree(Termlist,Tree),
      select_tuple(TS,Termlist,Atom,TSS),
      tss2forest(TSS,Forest3),
      select_tuples(TS,Forest2,Atom,Forest4),
      forest_union(Forest3,Forest4,Forest5),!.
select_tuples(TS,Forest,Atom,Forest)
   :- nl,write("fail select_tuples: "),
      nl,write(indent,"Termlist: ",TS),
      nl,write(indent,"Forest:   ",Forest),
      nl,write(indent,"Atom:     ",Atom),
      fail.


tss2forest([],[]) :- !.
tss2forest([TS|TSS],Forest2)
   :- Termlist_tree(TS,Tree),
      tss2forest(TSS,Forest1),
      insert_tree(Tree,Forest1,Forest2).

forest2tss([],[]) :- !.
forest2tss(Forest1,[TS|TSS])
   :- pick_tree(Forest1,Tree,Forest2),
      termlist_tree(TS,Tree),
      forest2tss(Forest2,TSS), !.
forest2tss(Forest,[])
   :- nl, write("fail forest2tss: "),
      nl, vis_forest(Forest), fail.




select_tuple(TS,TP,less_than(T1,T2),TSS)
   :- read_value(T1,TS,TP,V1),
      read_value(T2,TS,TP,V2),
      compare_values("<",V1,V2,TP,TSS),!.
select_tuple(TS,TP,less_equal(T1,T2),TSS)
   :- read_value(T1,TS,TP,V1),
      read_value(T2,TS,TP,V2),
      compare_values("<=",V1,V2,TP,TSS),!.
select_tuple(TS,TP,not_equal(T1,T2),TSS)
   :- read_value(T1,TS,TP,V1),
      read_value(T2,TS,TP,V2),
      compare_values("!=",V1,V2,TP,TSS),!.

select_tuple(TS,TP,equal(plus(T1,T2),T3),TSS)
   :- read_value(T1,TS,TP,V1),
      read_value(T2,TS,TP,V2),
      read_value(T3,TS,TP,V3),
      cases_values_plus(T1,T2,T3,V1,V2,V3,TS,TP,TSS),!.
select_tuple(TS,TP,equal(mult(T1,T2),T3),TSS)
   :- read_value(T1,TS,TP,V1),
      read_value(T2,TS,TP,V2),
      read_value(T3,TS,TP,V3),
      cases_values_mult(T1,T2,T3,V1,V2,V3,TS,TP,TSS),!.
select_tuple(TS,TP,equal(sign(T1),T3),TSS)
   :- read_value(T1,TS,TP,V1),
      read_value(T3,TS,TP,V3),
      cases_values_sign(T1,T3,V1,V3,TS,TP,TSS),!.
select_tuple(TS,TP,Atom,[])
   :- nl,write("fail select_tuple: "),
      nl,write(indent,"Termlist: ",TS),
      nl,write(indent,"Tuple:    ",TP),
      nl,write(indent,"Atom:     ",Atom),
      fail.



cases_values_plus(_,_,T3,V1,V2,V3,TS,TP,TSS)
   :- V1 = q(_,_,_), V2 = q(_,_,_),
      plus(V1,V2,V),
      write_value(T3,V3,V,TS,TP,TSS),!.
cases_values_plus(_,T2,_,V1,V2,V3,TS,TP,TSS)
   :- V1 = q(_,_,_),
      minus(V3,V1,V),
      write_value(T2,V2,V,TS,TP,TSS),!.
cases_values_plus(T1,_,_,V1,V2,V3,TS,TP,TSS)
   :- minus(V3,V2,V),
      write_value(T1,V1,V,TS,TP,TSS),!.

cases_values_mult(_,_,_,_,_,_,_,_,[]) :- trace(on), fail.
cases_values_mult(_,_,T3,V1,V2,V3,TS,TP,TSS)
   :- V1 = q(_,_,_), V2 = q(_,_,_),
      times(V1,V2,V),
      write_value(T3,V3,V,TS,TP,TSS),!.
```

```
cases_values_mult(_,T2,_,V1,V2,V3,TS,TP,TSS)
   :- V1 = q(_,_,_),
      divide(V3,V1,V),
      write_value(T2,V2,V,TS,TP,TSS),!.
cases_values_mult(T1,_,_,V1,V2,V3,TS,TP,TSS)
   :- divide(V3,V2,V),
      write_value(T1,V1,V,TS,TP,TSS),!.


cases_values_sign(_,T3,V1,V3,TS,TP,TSS)
   :- V1 = q(_,_,_),
      minus(q(1,0,1),V1,V),
      write_value(T3,V3,V,TS,TP,TSS),!.
cases_values_sign(T1,_,V1,V3,TS,TP,TSS)
   :- minus(q(1,0,1),V3,V),
      write_value(T1,V1,V,TS,TP,TSS),!.




compare_values("<",V1,V2,TP,[TP]) :- less_than(V1,V2),!.
compare_values("<=",V1,V2,TP,[TP]) :- less_equal(V1,V2),!.
compare_values("!=",V1,V2,TP,[TP]) :- not_equal(V1,V2),!.
compare_values(_,_,_,_,[]).



read_value(T,[T|_],[HDtuple|_],HDtuple) :- !.
read_value(T,[_|TLts],[_|TLtp],Value)
   :- read_value(T,TLts,TLtp,Value), !.
read_value(T,_,_,T).




write_value(_,_,_,[],_,[[]]) :- !.
write_value(T,V1,V2,[T|TLts],[_|TLtp],[TSS])
   :- V1 = anonymousvariable(),
      write_value(T,V1,V2,TLts,TLtp,TS),
      not(TS = []),
      TS = [TP], TSS = [V2|TP], !.
write_value(T,V1,V2,[HDts|TLts],[HDtp|TLtp],[TSS])
   :- V1 = anonymousvariable(),
      not(T=HDts),
      write_value(T,V1,V2,TLts,TLtp,TS),
      not(TS = []),
      TS = [TP], TSS = [HDtp|TP], !.
write_value(T,V,V,[T|TLts],[_|TLtp],[TSS])
   :- V = str(_),
      write_value(T,V,V,TLts,TLtp,TS),
      not(TS = []),
      TSS = [V|TLtp], !.
write_value(T,V,V,[HDts|TLts],[HDtp|TLtp],[TSS])
   :- V = str(_),
      not(T=HDts),
      write_value(T,V,V,TLts,TLtp,TS),
      not(TS = []),
      TS = [TP], TSS = [HDtp|TP], !.
write_value(T,V,V,[T|TLts],[_|TLtp],[TSS])
   :- V = q(_,_,_),
      write_value(T,V,V,TLts,TLtp,TS),
      not(TS = []),
      TSS = [V|TLtp], !.
write_value(T,V,V,[HDts|TLts],[HDtp|TLtp],[TSS])
   :- V = q(_,_,_),
      not(T=HDts),
      write_value(T,V,V,TLts,TLtp,TS),
      not(TS = []),
      TS = [TP], TSS = [HDtp|TP], !.
write_value(_,_,_,_,_,[]).


vis_relation(Rel)
   :- present(Rel),nl,
      write("*****",Rel),nl.

/*
vis_relation(r(ID,TS,Forest))
   :- write("relation(",ID,")"), vis_termlist(TS),
      vis_forest(Forest).
*/
vis_forest([]) :- !.
vis_forest(Forest1)
   :- pick_tree(Forest1,Tree,Forest2),
      termlist_tree(Termlist,Tree),
      vis_termlist(Termlist),
      vis_forest(Forest2).
```

G.7.1 INCLUDE atools.pro

```
predicates
  pick_tree (forest,tree,forest)
  insert_tree (tree,forest,forest)

  projectt  (string,Termlist,relation,relation)
    projectt1 (Termlist,Termlist,forest,forest)
    projectt2 (Termlist,Termlist,tree,tree)
    projectt3 (Expr,Termlist,tree,Expr)
    present (relation)
    present1 (Termlist,forest,integer)
    present2 (Termlist,tree,integer)


  Termlist_tree (Termlist,tree)

% load (string,state,relation)
/*
  match (string,Termlist,relation,relation)
     match2 (Termlist,forest,forest)


  bind (Termlist,tree,bindings,tree)
  bind2 (Termlist,Termlist)
*/

  invert (string,relation,relation)


  join (relation,relation,relation)
     cartecian_product (forest,forest,forest)
%     sort_join (Termlist,forest,Termlist,forest,forest)

     union_termlist (Termlist,Termlist,Termlist)
     cp1 (Termlist,forest,Termlist,forest,Termlist,forest)
     cp2 (Termlist,tree,Termlist,forest,Termlist,forest)
     cp3 (Termlist,tree,Termlist,tree,Termlist,forest)
     b (Substitution,Termlist,tree,Substitution)
     m (Substitution,Expr,Expr,Substitution)
     p (Termlist,Substitution,tree)


  diff (relation,relation,relation)
     intersection_list (Termlist,Termlist,Termlist)
     difference_list (Termlist,Termlist,Termlist)

     diff1 (Termlist,forest,Termlist,forest,forest)
     test (Expr,forest,forest)

  length (Depends,integer)
  length (Strings,integer)
  length (Termlist,integer)

  relation_union(relation,relation,relation)
  forest_union (forest,forest,forest)


clauses
  projectt (HEAD,MTermlist,r(_,Termlist,Forest1),
   r(HEAD,MTermlist,Forest2))
     :- projectt1(MTermlist,Termlist,Forest1,Forest2).

  projectt1 (I1,I2,S1,S4)
     :- pick_tree(S1,T1,S2),
        projectt2(I1,I2,T1,T2),
        projectt1(I1,I2,S2,S3),!,
        insert_tree(T2,S3,S4), !.
  projectt1 (_,_,[],[]).

  projectt2 ([X|I1],I2,T,t(C,[T2]))
     :- projectt3(X,I2,T,C),
        projectt2(I1,I2,T,T2), !.
  projectt2 ([],_,_,leaf).

  projectt3 (anonymousvariable(),_,_,anonymousvariable()):- !.
  projectt3 (str(X),_,_,str(X)) :- !.
  projectt3 (q(S,N,D),_,_,q(S,N,D)) :- !.
  projectt3 (var(X),[var(X)|_],t(C,_),C) :- !.
  projectt3 (X,[_|Y],t(_,[R]),C) :- projectt3(X,Y,R,C).



  pick_tree ([leaf],leaf,[]) :- !.
  pick_tree ([t(A,S)|R],t(A,[T]),R)
     :- pick_tree(S,T,[]), !.
  pick_tree ([t(A,S)|R],t(A,[T]),[t(A,R1)|R])
     :- pick_tree(S,T,R1), !.


  insert_tree (T,[],[T]) :- !.
  insert_tree (leaf,[leaf],[leaf]) :- !.
  insert_tree (t(A,[T]),[t(A,S1)|R],[t(A,S2)|R])
     :- insert_tree(T,S1,S2), !.
  insert_tree (t(A1,S1),[t(A2,S2)|R],[t(A1,S1),t(A2,S2)|R])
     :- less_than(A1,A2), !.
  insert_tree (t(A1,S1),[t(A2,S2)|R],[t(A2,S2)|R1])
     :- less_than(A2,A1), insert_tree(t(A1,S1),R,R1), !.


  present (r(ID,I,S)) :- nl,write(indent,ID,indent),present1(I,S,0),nl.

  present1(_,[],0) :- write("NO"), nl,  !.
  present1(_,[leaf],0) :- write("YES"),  nl, !.
  present1(_,[],1) :- nl, write("One solution."), nl,  !.
  present1(_,[],L) :- nl, write(L," solutions."), nl, !.
  present1(I, S,L)
     :- pick_tree (S,Solution,R),
        nl, present2 (I,Solution,C), L1 = L+C,
        present1(I,R,L1).
```

```
present2 ([X],    t(anonymousvariable,[leaf]),1)
   :- vis_term(X), write("=_."), !.
present2 ([X|R],  t(anonymousvariable,[T]),I)
   :- vis_term(X), write("=_, "), present2(R,T,I), !.
present2 ([X],    t(Y,[leaf]),1)
   :- vis_term(X), write("="), vis_term(Y), write("."), !.
present2 ([X|R],  t(Y,[T]),I)
   :- vis_term(X), write("="), vis_term(Y), write(", "),
      present2(R,T,I), !.

Termlist_tree ([Expr|Rest],t(Expr,[T]))
   :- Termlist_tree(Rest,T), !.
Termlist_tree ([],leaf).


load (S,[Relation|_],Relation)  :- Relation = r(S,_,_), !.
load (S,[_|RestState],Relation) :- load(S,RestState,Relation), !.
load (S,[],r(S,[],[])).

invert(HEAD,r(_,_,[]),r(HEAD,[],[leaf])) :-!.
invert(HEAD,r(_,_,_),r(HEAD,[],[])) :-!.


   cartecian_product ([],_,[]) :- !.
   cartecian_product (_,[],[]) :- !.
   cartecian_product ([t(S,T1)|R1],Forest,[t(S,T2)|R2])
      :- cartecian_product (T1,Forest,T2),
         cartecian_product (R1,Forest,R2), !.
   cartecian_product ([leaf],Forest,Forest) :-!.

join(r(ID,Termlist1,Forest1),r(_,Termlist2,Forest2),
 r(ID,UTermlist,UForest))
   :- %card(Termlist1,N), card(Termlist2,N),!,
      union_termlist(Termlist1,Termlist2,UTermlist),
      cp1(Termlist1,Forest1,Termlist2,Forest2,UTermlist,UForest),!.
join(Rel1,Rel2,Rel1)
   :- nl, write("fail join: "),
    nl, write(indent,"Rel1: ",Rel1),
    nl, write(indent,"Rel2: ",Rel2), readchar(_), fail.

union_termlist([],[],[]) :- !.
union_termlist([],[T1|TS1],TS3)
   :- union_termlist([],TS1,TS2),
      insert(T1,TS2,TS3), !.
union_termlist(TS1,TS2,TS4)
   :- append(TS1,TS2,TS3),
      union_termlist([],TS3,TS4).


cp1(TS1,Forest1,TS2,Forest2,UTS,UForest)
   :- pick_tree(Forest1,Tree1,Forest3),
      cp2(TS1,Tree1,TS2,Forest2,UTS,UForest1),
      cp1(TS1,Forest3,TS2,Forest2,UTS,UForest2),
      forest_union(UForest1,UForest2,UForest), !.
cp1(_,[],_,_,_,[]).

cp2(TS1,Tree1,TS2,Forest2,UTS,UForest)
   :- pick_tree(Forest2,Tree2,Forest3),
      cp3(TS1,Tree1,TS2,Tree2,UTS,UForest1),
      cp2(TS1,Tree1,TS2,Forest3,UTS,UForest2),
      forest_union(UForest1,UForest2,UForest),!.
cp2(_,_,_,[],_,[]).

cp3(TS1,Tree1,TS2,Tree2,UTS,[UTree])
   :- b([],TS1,Tree1,B1),
      b(B1,TS2,Tree2,B2),
      p(UTS,B2,UTree),!.
cp3(_,_,_,_,_,[]).


b(B1,[T|TS],t(E,[Tree]),B3)
   :- m(B1,T,E,B2),
      b(B2,TS,Tree,B3), !.
b(B,[],leaf,B).

m(B,T,E,B)
   :- member(binding(T,E),B), !.
m(B1,T,E,[binding(T,E)|B2])
   :- member(binding(T,anonymousvariable),B1),
      remove(binding(T,anonymousvariable),B1,B2),!.
m(B,T,E,[binding(T,E)|B])
   :- not(member(binding(T,_),B)).

p([],_,leaf) :- !.
p([T|TS],B,t(E,[Tree]))
   :- member(binding(T,E),B),!,
      p(TS,B,Tree).


diff(r(S,Termlist1,Forest1),r(_,Termlist2,Forest2),
r(S,Termlist5,Forest6))
   :- intersection_list(Termlist1,Termlist2,Termlist3),
      difference_list(Termlist1,Termlist3,Termlist4),
      append(Termlist3,Termlist4,Termlist5),
      projectt1(Termlist3,Termlist2,Forest2,Forest3),
      projectt1(Termlist5,Termlist1,Forest1,Forest5),
      diff1(Termlist5,Forest5,Termlist3,Forest3,Forest6).



   intersection_list([A|R1],T,[A|R2])
      :- member(A,T),
          intersection_list(R1,T,R2), !.
      intersection_list([_|R1],T,R2)
         :- intersection_list(R1,T,R2), !.
      intersection_list([],_,[]).

      difference_list([A|R1],T,R2)
         :- member(A,T),
            difference_list(R1,T,R2), !.
      difference_list([A|R1],T,[A|R2])
         :- difference_list(R1,T,R2), !.
      difference_list([],_,[]).



  diff1(Termlist1,Forest1,Termlist2,Forest2,[T1|Forest3])
     :- Forest1  = [T1|S1], T1 = t(N1,_),
        Forest2  = [t(N2,_)|_],
        less_than(N1,N2),
        diff1(Termlist1,S1,Termlist2,Forest2,Forest3), !.

  diff1(Termlist1,Forest1,Termlist2,Forest2,Forest5)
     :- Termlist1 = [_|RV1],
        Forest1 = [t(N,US1)|S1],
        Termlist2 = [_|RV2],
        Forest2 = [t(N,US2)|S2],
        diff1(RV1,US1,RV2,US2,Forest3),
        diff1(Termlist1,S1,Termlist2,S2,Forest4),
test(N,Forest3,T),
        append(T,Forest4,Forest5), !.


  diff1(Termlist1,Forest1,Termlist2,Forest2,Forest3)
     :- Forest1  = [t(N1,_)|_],
        Forest2  = [t(N2,_)|S2],
        less_than(N2,N1),
        diff1(Termlist1,Forest1,Termlist2,S2,Forest3), !.


  diff1(_,[],_,_,[]) :- !.
  diff1(_,_,[],[leaf],[]) :- !.
  diff1(_,Forest,_,[],Forest) :- !.

  test(_,[],[]) :- !.
  test(N,SK,[t(N,SK)]).


length([],0) :- !.
length([_|R],N1) :- length(R,N), N1 = N+1.
```

```
  relation_union(Rel1,Rel2,Rel3)
     :- Rel1 = r(S,Index1,Forest1),
        Rel2 = r(S,Index2,Forest2),
        card(Index1,N), card(Index2,N),!,
        forest_union(Forest1,Forest2,Forest3),
        Rel3 = r(S,Index1,Forest3),
     % nl, write("SUCCES relation_union: "),
     % nl, write(indent,"Rel1: "), present(Rel1),
     % nl, write(indent,"Rel2: "), present(Rel2),
     % nl, write(indent,"Rel3: "), present(Rel3),
      !.
  relation_union(Rel1,Rel2,Rel1)
     :- nl, write("fail relation_union: "),
      nl, write(indent,"Rel1: "), present(Rel1),
      nl, write(indent,"Rel2: "), present(Rel2), readchar(_), fail.


  forest_union ([],Forest,Forest) :- !.
  forest_union (Forest1,Forest2,Forest5)
     :- pick_tree(Forest1,Tree1,Forest3),
        forest_union(Forest3,Forest2,Forest4),
insert_tree(Tree1,Forest4,Forest5).
```

Appendix H

Last Page