ProQuel: Using Prolog to implement a deductive database system

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Informationssysteme

Jan Burse

ProQuel: Using Prolog to Implement a Deductive Database System

June 1992

177

ETH Zürich
Departement Informatik
Institut für Informationssysteme
Prof. Dr. Robert Marti

Authors' address:

Institut für Informationssysteme
ETH Zentrum, CH-8092 Zurich, Switzerland

e-mail: burse@inf.ethz.ch

Abstract

The ProQuel system, presented in this report, is an outgrowth of the LogiQuel project [MWW89] which was started as a vehicle for research in logic based query languages. During the progress of the project the need to settle on a cleaned up version of a deductive database system kernel became clear and the ProQuel system was initiated. ProQuel has been implemented in Prolog, in contrast to its predecessors, which were implemented in Modula-2. The implementation language Prolog was chosen because of its symbolic manipulation capabilities. As it turned out, many of the used algorithms could be written much more easily and concisely in Prolog. Moreover we believe that later on the use of Prolog will help in the faster development of prototypes that explore new concepts.

The report starts with an informal presentation of the ProQuel query language. It then documents an implementation that is strictly guided by the clear separation between database target dependent and independent code. At the end, a brief description of the theoretical framework is given that was used as a basis for the ProQuel language.

Contents

Chapter 1

Introduction

A deductive database system consists of an inference engine and a knowledge base. When a query is posed to the knowledge base, the inference engine has to deduce the answer. The queries and the knowledge base are represented in a formal language. The inference engine puts the syntactic objects queries and knowledge bases into relation. The relation is defined as follows: If the answer to the closed query Q posed to the knowledge base D is yes, then we write:

$$D \mid\!\vdash Q$$

In general an inference engine $\mid\!\vdash$ may have some unexpected properties. For example there might be a knowledge base D and a query Q such that $D \mid\!\vdash Q$ and $D \mid\!\vdash \neg Q$. In deductive databases such a behaviour of the inference engine is not desired because a consistent description of the problem space is assumed. Beneath consistency one also wants to have completeness. The desired properties of $\mid\!\vdash$ are given in the following:

- Consistency: There is no Q with $D \mid\!\vdash Q$ and $D \mid\!\vdash \neg Q$
- Completeness: For all Q it is $D \mid\!\vdash Q$ or $D \mid\!\vdash \neg Q$

On a first attempt one might wish to use predicate calculus as a basis for the inference engine. This is done in the *proof theoretic* approach which means that knowledge bases are first order theories and the predicate calculus $\vdash$ is used for the inference relation $\mid\!\vdash$. Unfortunately the proof theoretic approach has none of the desired properties in general. The unrestricted use of first order theories may lead to inconsistencies and incompleteness. Take for example the theory $D = \{p(a), \neg p(a)\}$, we have $D \vdash p(a)$ and $D \vdash \neg p(a)$. It follows that consistency is not guaranteed in general. Or take for example the empty theory $D = \emptyset$, we have $D \nvdash p(a)$ and $D \nvdash \neg p(a)$. Hence completeness is not guaranteed either in general.

We have seen that the proof theoretic approach is not suitable for deductive databases because none of the desired properties holds in general. Therefore we have to look for another approach to define a inference relation $\mid\!\vdash$ for deductive databases. In ProQuel the *model theoretic* approach is taken which means that a first order structure M_D is assigned to every knowledge base D. The answer to a query Q is then obtained by checking the satisfiability in M_D:

$$M_D \models Q$$

By definition, the model theoretic semantic clearly assures consistency and completeness. The problem that now arises is the choice of the appropriate model. In ProQuel the *standard model* for *stratified rules* is used. Stratified rules allow a restricted form of negation in the presence of recursion. The choice of the standard model can be uniquely characterized by the choice of a minimal and preserving model for the completion of the rules (see section 5.3).

In general the standard model of stratified rules cannot be computed because the true sentences are neither decidable nor semi decidable. Therefore a fragment of the rules and queries has to be considered to allow a feasible implementation. In ProQuel we have restricted ourselves to *allowed* rules and queries. Allowedness assures the computational tractability of queries if a standard model with finite predicates exists (see section 5.4).

1.1 Report Organization

The report is organized as follows. In chapter 2, the syntax and behaviour of the ProQuel language are informally presented. The chapter is intended to suit users of the ProQuel system that have a running system at hand. Then chapter 3 and 4 present the implementation of the ProQuel language and demonstrate the use of Prolog. The reader of these chapters should be familiar with Prolog and it is recommended to have the source code of the ProQuel system at hand. Finally in chapter 5 the theoretical background for the ProQuel implementation is given. For this chapter it is advantageous to have knowledge of deductive database theory [CGT90], [Llo87, p. 142-171].

1.2 Acknowledgements

I would like to thank my supervisor, Robert Marti, for his steady support of this work. I am in great debt to Mike Boehlen and Roman Gross for their valuable discussions and their contributions to the code. I would also like to thank Renaud Hirsch who had always time to listen to my ideas. Finally am grateful to Angelika Dittrich, Duri Schmid and Mitch Wyle for their helpful comments and corrections on earlier draft.

Chapter 2

ProQuel Syntax

In the following, the ProQuel language is informally presented. There will be no discussion of the concepts and methods involved. The emphasis is on the syntactic nature of the commands and their intended meaning. The presentation starts with the lexical units of the language in section 2.1. The ProQuel language is then divided into the data retrieval, data definition and data manipulation commands which are presented in sections 2.2, 2.3 and 2.4 respectively. Finally, in section 2.5, a small example is given that illustrates the use of the ProQuel language.

The EBNF-notation is used in this chapter to define the lexical units and commands. It consists of identifiers as non-terminals, strings in quotes (") as terminals and the constructs A|B (alternative), [A] (option) and {A} (repetition).

2.1 Lexical Units

The current ProQuel implementation consists of a primitive command line interface which performs a read-execute loop. The incoming string is split into symbols. Blanks and comments between `/*` and `*/` are skipped. The following symbols are distinguished:

```
var       = capital { letter | digit }.
ident     = non-capital { letter | digit }.
str       = "'" { char | "''" } "'".
int       = digit { digit }.
float     = int "." int.
```

Examples:

`Salary` is a variable (var)
`employee` is an identifier (ident)
`'Don''t worry'` is a character string (str)
`123` is an integer number (int)
`12.34` is a floating point number (float)

Some combinations of special characters and some reserved words are also recognized as independent symbols. Reserved words cannot be used in the role of identifiers.

The following character combinations and reserved words are recognized:

```
<-      ->      |       &       ~       @       #
=       \=      <       <=      >       >=      -
+       *       /       (       ,       )

assert  clear   create  div     drop    float   int
list    mod     query   retract str     true    quit
```

The character combinations `->`, `|`, `&`, `~`, `@`, `#`, `<=`, `>=`, `\=` are used for the mathematical symbols $\rightarrow$, $\vee$, $\wedge$, $\neg$, $\forall$, $\exists$, $\leq$, $\geq$, $\neq$ respectively as these symbols are not found on common keyboards. Except for this chapter the mathematical symbols are used in the text.

2.2 Data Retrieval

The sorts of ProQuel are integers, floats and strings. Expressions are used to denote the application of predefined operations on constants and current values of variables. There are constants for all the values of each sort. A variable is assumed to have a current value of one sort. The predefined operations are the arithmetic operations on integers and floats.

```
expr        = ["-"] term { ("+"|"-") term }.
term        = factor { ("*"|"/"|"mod"|"div") factor }.
factor      = "(" expr ")" | str | int | float | var.
```

Example:

`3*(Salary+100)` is an expression (expr)

The ProQuel queries are simply formulas. A formula combines user-defined and built-in predicates by logical connectives. A formula may contain quantifiers and negation. The built-in predicates are comparisons which are allowed between expressions of the same sort.

```
form        = disj [ "->" disj ].
disj        = conj { "|" conj }.
conj        = quan { "&" quan }.
quan        = { "~" | ("@"|"#") var } simp.
simp        = "(" form ")" | "true" | comp | atom.
comp        = expr ("="|"<"|">"|"<="|">="|"\=") expr.
atom        = ident [ "(" expr { "," expr } ")" ].
```

Example:

`employee(Name,Sal) & Sal>=4000` is a formula (form)

A query is posed by the query command. A closed query is answered by **yes** or **no**. A query with variables occurring free is answered by giving the set of substitutions that satisfy the query.

```
query         = "query" form
```

Before accepting a query, various checks and transformation are performed. First the sorts of the query are deduced, then the query is normalized and finally the allowedness of the query (see section 5.4) is checked. If one of these checks fails, the query will not be accepted. Otherwise a plan is generated, the needed tables are materialized and the query is evaluated. For more details on the implementation see section 3.4, 3.5, 3.6, 3.8, 3.9.

2.3 Data Definition

The `create`, `drop`, `clear` and `list` commands are concerned with the definition of predicates. Every predicate has to be declared by a `create` command before it can be used in a rule or query. A rule has the form $P \leftarrow A$ where P is a predicate atom and A is a formula. A rule is said to belong to the predicate that forms the head of the rule. The `drop` command removes all the rules that belong to the specified predicate and its predicate definition. The `clear` only removes the rules and leaves the predicate definition intact. The `list` command displays the current predicate definitions.

```
create        = "create" ident [ "(" sort { "," sort } ")" ].
drop          = "drop" ident.
clear         = "clear" ident.
list          = "list".
sort          = "int" | "float" | "str".
```

The `create` command specifies the number of arguments and the corresponding sorts of the arguments for a predicate name. It is not allowed to use the same predicate name with different numbers of arguments or to redefine a predicate name. If you want to redefine a predicate, you will have to drop it first.

2.4 Data Manipulation

A ProQuel knowledge base consists of a set of rules only. Facts are regarded as rules of the form $P \leftarrow true$ and are abbreviated by P alone. The rules of the knowledge base are handled by the `assert`, `retract` and `show` commands. The `assert` command adds a rule to the knowledge base, the `retract` command removes a rule from the knowledge base. The `show` command displays the current rules that belong to the specified predicate.

```
assert        = "assert" rule
retract       = "retract" rule
show          = "list" ident
rule          = atom [ "<-" form ]
```

Rules are stored as a sequence of tokens. Therefore rules that differ only in a renaming of the variables are considered to be different. Before accepting a rule in

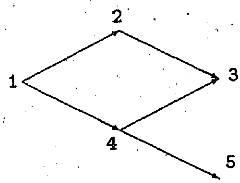

Figure 2.1: Example Computer Network

an `assert` command, various checks and transformations are performed. First the sorts of the variables in the rule are deduced, then the rule is normalized and finally the allowedness of the rule (see section 5.4) and the stratification of the knowledge base (see section 5.3) is checked. The rule will be stored if none of these checks fails. For more details on the implementation see section 3.4, 3.5, 3.6, 3.7.

2.5 Example

To illustrate the power of the ProQuel language an examples database to solve a small problem is given. Its formulation is very short and elegant in comparison to a formulation in relational algebra or SQL. It is a toy example that shows the use of the most fundamental and prominent features of ProQuel, namely *recursion*, *negation* and *quantifiers*. The example is essentially taken from [CGT90, p.217]. In the following a greater sign (>) marks the input send to ProQuel.

Consider the computer network depicted in figure 2.1. Two computers are connected if and only if there are one or more links between them. Two computers are said to be safely connected if and only if there is still a new connection when any online computer fails. The computer network is represented by `host` and `link` facts:

```
>create host(int)
>assert host(1)
>assert host(2)
>assert host(3)
>assert host(4)
>assert host(4)

>create link(/*from*/int,/*to*/int)
>assert link(1,2)
>assert link(2,3)
>assert link(1,4)
>assert link(4,3)
>assert link(4,5)
```

It is now easy to describe all the connections by a new predicate `connected`. The predicate `connected` simply needs two rules that define the transitive closure of `link` by recursion:

```
>create connected(/*from*/int,/*to*/int)
>assert connected(X,Y) <- link(X,Y)
>assert connected(X,Y) <- connected(X,Z) & connected(Z,Y)
```

The key in solving the problem is a new predicate `circumvent`. The intention is that `circumvent(X,Y,Z)` holds whenever there is a connection from `Y` to `Z` without passing through `X`. A new predicate `safe` may then define the safe connections in terms of `circumvent` and `connected`:

```
>create circumvent(/*without*/int,/*from*/int,/*to*/int)
>assert circumvent(X,Y,Z) <- host(X) & link(Y,Z) & X\=Y & X\=Z
>assert circumvent(X,Y,Z) <- circumvent(X,Y,H) & circumvent(X,H,Z)

>create safe(int,int)
>assert safe(X,Y) <- connected(X,Y) &
    @Z(host(Z) & Z\=X & Z\=Y -> circumvent(Z,X,Y))
```

So the problem is captured by allowed rules that are stratified. Hence the knowledge base is accepted by the the ProQuel system and the user may now issue the following queries:

```
>query safe(1,5)
no
>query safe(1,3)
yes
>query safe(1,X)
   X
----
   3
   2
   4
>query connected(1,X) & ~safe(1,X)
   X
----
   5
```

Chapter 3

ProQuel Core

The core is the target independent part of the ProQuel code. It is entirely written in SICStus Prolog and it always remains the same independent of the current target. The chapter examines the different parts of the core which are subsequently described in the following sections. The order of the sections reflects the execution flow of the ProQuel core. From time to time, code extracts of the core are used to illustrate the use of Prolog in this project. Target relevant information is found in the next chapter.

3.1 Using Prolog

ProQuel features the use of Prolog as an implementation language. Prolog has its origins in the early 70's when Kowalski postulated $Algorithm = Logic + Control$. At the same time Colmerauer et al. at the University of Marseilles-Aix developed a theorem prover, written in Fortran, which they used to implement natural language processing systems. The theorem prover was called Prolog (Programmation en Logique) and embodied the idea of Kowalski. It was the advent of the new programming language Prolog. The first Prolog compiler was then developed by Warren et al. in the late 70's. Today there are numerous commercially available Prolog implementations. The language itself has a de facto standard, the Edinburgh Prolog family [CM84, SS86].

In pure Prolog, the notions of declaration, data type and module do not exist. There are some Prolog implementations with such extensions but they are not yet widespread enough. Thus it was in our own responsibility to create structured and self documenting programs. We used the standard possibility to distribute the program over different files. By using comments we tried to exhibit the module structure of these files. The number of clauses to implement a certain predicate is often very small in Prolog. The image is reversed in comparison to a procedural language like C, where a simple symbolic manipulation often needs a great number of statements. This due to the capability of Prolog to match and automatically construct terms by unification. Therefore ProQuel consists of few modules that are rather short and that contain a large number of exported predicates (see figure 3.1).

Part	Module	Lines	Predicates	Clauses
Core	main.p	293	24	43
	parser.p	395	58	186
	checker.p	277	21	91
	error.p	98	7	28
SQL Target	meta.p	281	39	79
	eval.p	138	14	31
POSTQUEL Target	meta.p	248	38	71
	eval.p	149	15	28
Prolog Target	meta.p	31	4	3
	eval.p	140	15	31

Figure 3.1: Module Statistics

The rewriting of the original Modula-2 program to Prolog has considerably improved the quality of the code. Clearly a rewriting always improves the code and other languages are equally well suited to implement a deductive database systems. Nevertheless Prolog has been chosen and some Prolog concepts have proven well-suited to the implementation of a deductive database systems. For example the built-in *definite clause grammars* of Prolog were used whenever a parser or unparser was needed (see section 3.3). *Unification* was very useful in the sort inference (see section 3.4) whereas *backtracking* was needed in the normalization (see section 3.5). The other parts of the core also took advantage of Prolog as the sparcity of code shows. Finally, Prolog was useful in the implementation of the Prolog target where in particular *negation as failure* was used to implement the ProQuel negation (see section 4.4).

3.2 Separation Core and Target

One of the key goals in the design of the ProQuel system was the support of different database targets. The database targets are responsible for the *data storage* and the *query evaluation*. They are usually commercially available relational database systems. The ProQuel system divides into the database target independent part and the database target dependent parts. The target independent part is called the core. It is entirely written in SICStus Prolog and remains the same code for every target. It has a fixed interface to the target dependent part. The target dependent part has to be re-implemented for every new database target. It is not necessarily completely written in Prolog and it may differ considerably from target to target.

Figure 3.2 shows the processing of queries and rules in the core. Queries and rules are both first parsed and converted into Prolog terms (see section 3.3). The Prolog terms are then given to the sort inference to deduce the missing sorts (see section 3.4). If this check was successful, the Prolog terms are normalized (see section 3.5). The normal forms must then pass the allowedness check (see section 3.6). If this

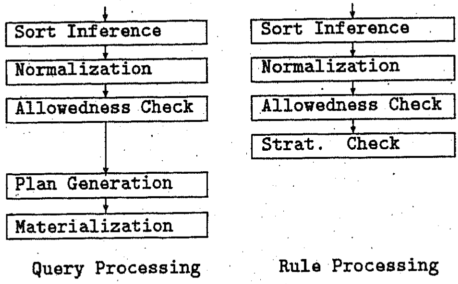

Figure 3.2: Processing of Queries and Rules

check was successful, rules and queries go different ways. For rules the stratification of the whole knowledge base is checked (see section 3.7). If this was successful, rules are stored for later use in the evaluation of queries. For queries an evaluation plan is generated (see section 3.8). Then all relevant tables are materialized according to the evaluation plan (see section 3.9) and the query is evaluated.

There is a great benefit in the separation of the code into the core and the target. Porting ProQuel from one system to another becomes much easier because it can be expected that only the target has to be changed. That ProQuel now runs on different targets also allows comparisons between different targets. Designing the core was not simple. A clear, versatile and simple interface to potential targets had to be designed. And, very important, the core should solve all the problems that are target independent in advance. In fact, figure 3.2 shows an analysis of the different tasks that a deductive database system must carry out independent of the target database.

3.3 Scanner, Parser and Unparser

To develop scanners, parsers and unparsers, Prolog provides a simple and elegant tool in the form of definite clause grammars (DCGs). The notation of DCGs is built around BNF or context free grammar rules augmented by semantic actions and attributes [CM84]. The specification and implementation of DCGs are done through a rewriting to Prolog code by the Prolog system itself. Hence DCGs are ready to use and easy to understand.

The first place where DCGs were used is the scanner. The scanner converts a stream of characters into a stream of tokens. The realization by means of DCGs is straight forward. Figure 3.3 shows a code fragment of the scanner. The next place where DGCs were used is the parser. The parser converts a stream of tokens into a Prolog term. Figure 3.4 shows the code fragment of the parser that is used to parse expressions. The repeating occurrence of the nonterminal `term` in the

```
%% tokens(-tokenlist)
tokens(Ts) --> " ", tokens(Ts).
tokens([T|Ts]) --> token(T), tokens(Ts).
tokens([]) --> "".

%% token(-token)
token(plus) --> "+".
token(minus) --> "-".
...
```

Figure 3.3: Code Fragment Scanner

```
%% expr(-term)
expr(Y) --> [minus], !, term(X), rest(neg(X),Y).
expr(Y) --> term(X), rest(X,Y).

%% rest(+term,-term)
rest(X,Z) --> [minus], !, term(Y), rest(sub(X,Y),Z).
rest(X,Z) --> [plus], !, term(Y), rest(add(X,Y),Z).
rest(X,X) --> [].

...
```

Figure 3.4: Code Fragment Parser

original EBNF production of expr is realized by tail recursion. The last place where DCGs were used is the unparser. The unparser converts a Prolog term back to a character stream. It has to be implemented separately because the scanner and parser, although programmed in Prolog, can not be used bidirectionally. The implementation of the unparser is straight forward. Figure 3.5 shows a code fragment of the unparser.

For the parser some additional comments are appropriate. In general, backtracking is allowed in DCGs but it has to be avoided for efficiency reasons. The original grammar of ProQuel needs backtracking to be parsed correctly because it is not

```
%% unparseform(+term)
unparseform(imp(X,Y)) --> !, unparsedisj(X), " -> ", unparsedisj(Y).
unparseform(X) --> unparsedisj(X).

%% unparsedisj(+term)
unparsedisj(or(X,Y)) --> !, unparsedisj(X), " | ", unparseconj(Y).
unparsedisj(X) --> unparseconj(X).

...
```

Figure 3.5: Code Fragment Unparser

LL1 [ASU86]. To overcome the problem a weaker grammar that can be parsed without backtracking was implemented. As a result there can be sentences that are not correct but accepted by the weaker grammar. These sentences have to be singled out later on by the sort inference. For example the input `(1<2)=3` would be accepted by the weaker grammar but rejected by the sort inference because 3 is of sort `int` and `(1<2)` is a formula.

3.4 Sort Inference

The usage of the original many sorted language would be very tedious. To be more user friendly the queries and rules are sort free. The user does not have to specify the sorts of the variables and he can use the same symbol for different related built-ins. The idea is that a many sorted formula is reconstructed from the original input. Consider for example the following formula:

$$p(X) \wedge Y = X.$$

Assume that p has sort *int* in its argument, then there is exactly one way to interpret the input as a many sorted query. The $=$ symbol must denote equivalence between integers and the variables must belong to V_int which we will be denoted by X^{int} and Y^{int}. The corresponding many sorted formula is:

$$p(X^{int}) \wedge Y^{int} =_{int} X^{int}$$

That the same symbol of the sort free language may stand for different symbols of the many sorted language is called *overloading*. For example the $+$ symbol may stand for $+_1 : int \times int \rightarrow int$ and $+_2 : date \times int \rightarrow date$. A special form of overloading is *polymorphism* where the overloading takes place for some parameters that range over the whole set of sorts. Assume for example that for every sort $s \in S$ there is a sort $list(s) \in S$ that denotes lists of elements of sort s. The symbol *member* may then stand for the symbols $member_s$ with $member_s \subseteq s \times list(s)$. An other example of polymorphism is the overloading of the $=$ symbol.

In principle any form of overloading is allowed in ProQuel because the sort inference is regarded as a pre-process that is not part of the semantic. In general, the presence of overloading may lead to ambiguities of queries and more severe problems in the meaning of rules. Therefore we have avoided overloading in general and restricted ourselves to polymorphism of some built-ins and functions. One advantage is that polymorphism leads only to one form of ambiguities, namely the under determinism of sorts. Another advantage is that only unification and no backtracking is needed in the sort inference process.

To simplify matters, formulas and expressions are handled in the same way. For this purpose the sort descriptors of figure 3.6 are introduced. Formulas and expressions are separated by the descriptors `f` and `e(_)`. There is an additional distinction between string expressions and numeric expressions. The numeric expressions are further divided into integer expressions and float expressions. For example the

```
-                 term
├─f               formula
└─e(_)            expression
   ├─e(s)         string expression (str)
   └─e(n(_))      numeric expression
       ├─ e(n(i)) integer expression (int)
       └─ e(n(f)) float expression (float)
```

Figure 3.6: Sort Descriptors

```
%% chk(+term,-sort)
chk(and(X,Y),f) :- chk(X,f), chk(Y,f).
chk(eq(X,Y),f) :- chk(X,e(S)), chk(Y,e(S)).
chk(var(X),e(S)).
chk(p(X),f) :- chk(X,e(n(i))).
...

?- chk(and(p(var('X')),eq(var('Y'),var('X'))),f).
yes
```

Figure 3.7: Code Fragment Sort Inference

descriptor pattern `e(n(_))` may stand for the descriptors `e(n(i))` or `e(n(f))`, i.e. for float or integer expressions.

The predicate chk/2 is used for the sort inference. Figure 3.7 shows a code fragment of the sort inference. The fragment shows the rules for the conjunction, for the equivalence and for variables. It also shows the rule for the user defined predicate p which is to be constructed from the predicate definition of p. Finally, the application of the sort predicate to $p(X) \land Y = X$ is also shown.

The method presented so far is easily extended to consistently support variables and quantifiers. An appropriate extension is implemented and used in the ProQuel core. It is also understandable that the term sort inference instead of the term type checking is used to emphasize the synthesis of sort descriptors. Type checking should refer to programming languages with strong typing where every variable has to be supplied by a declared type. Note also that the sort free language is rich enough to describe every allowed many sorted formula. Hence there is no possibility of unsolvable ambiguities in the formulation of queries and rules.

3.5 Normalization

Normalization is an important step in the processing of queries and rules. It is used to transform a query or rule into a form that is more suitable for further processing. The way normalization is done not only determines the set of accepted queries and rules, it also has an impact on the performance of the query evaluation. To get an

$$
\begin{array}{llll}
A \to B & \mapsto & \neg A \lor B & (R1) \\
\forall x\, A & \mapsto & \neg \exists x\, \neg A & (R2) \\
 & & & \\
\neg(A \land B) & \mapsto & \neg A \lor \neg B & (R3) \\
\neg(A \lor B) & \mapsto & \neg A \land \neg B & (R4) \\
\neg\neg A & \mapsto & A & (R5) \\
 & & & \\
A \land (B \lor C) & \mapsto & (A \land B) \lor (A \land C) & (R6) \\
(A \lor B) \land C & \mapsto & (A \land C) \lor (B \land C) & (R7) \\
\exists x\, (A \lor B) & \mapsto & \exists x\, A \lor \exists x\, B & (R8)
\end{array}
$$

Table 3.1: Rewrite Rules

impression of the normalization problem consider the following query:

$$\forall X(p(X) \to X < 7 \land X > 3)$$

The normal form that ProQuel generates is as follows:

$$\neg\exists X(p(X) \land X \geq 7) \land \neg\exists X(p(X) \land X \leq 3)$$

Assume that we want a translation of the query to SQL. From the original query an appropriate translation to SQL is not easy to see. On the other hand the normal form contains only the connectives $\land$ and $\neg\exists$. These connectives directly correspond to joins and NOT EXISTS sub-queries in the SQL world.

Normalization can vary in the technique that is used to obtain the normal form and in the structure of the normal form. In LogiQuel [MWW89] the formulas were quantifier free and rather bulky Modula-2 program was used to obtain the *disjunctive normal form* of a query. In NewQuel [Bur89] the *combined normal form* was introduced because quantifiers were now present and the disjunctive normal form was no longer suitable. The combined normal form is essentially derived from the transformation rules in [Llo87, p.113]. The combined normal form was constructed by a rather bulky Modula-2 program as well.

The normalization of the previous projects was not satisfying. The programs to obtain the normal form were large and difficult to maintain and the normal form was still not perfectly suited for further processing. Therefore in ProQuel [Hel91] a new approach was taken. The new normal form is called the *or-free parts normal form* and it is now obtained with the help of a general *narrowing system* written in Prolog. Table 3.1 shows the rewrite rules that are supplied to the narrowing system to obtain the or-free parts normal form.

Every derivation by the rules (R1) to (R8) terminates with a formula of the form $N_1 \lor \ldots \lor N_n$ that is logically equivalent to the original formula. The N_i are called the or-free parts because they are always free of disjunctions ($\lor$). The advantage of the rules (R1) to (R8) is that they are able to remove disjunctions from the inside of any formula even in the presence of quantifiers without the introduction of new

```
%% +term ==> -term
imp(X,Y) ==> or(not(X),Y).
all(V,X) ==> not(ex(V,not(X))).
not(and(X,Y)) ==> or(not(X),not(Y)).
...

%% try(+term,-term)
try(X,Y) :- X==>Y.
try(X,Y) :- get(X,N,A), try(A,B), put(X,N,B,Y).

%% norm(+term,-term)
norm(X,Y) :- try(X,Z), !, norm(Z,Y).
norm(X,X).

?- norm(not(and(p,or(q,r))),X)
X=or(not(p),and(not(q),not(r)))
```

Figure 3.8: Code Fragment Normalizer

complexities in the form of additional negations.

The or-free parts normal form is *not uniquely determined.* For example the formula $\neg(p \wedge (q \vee r))$ has the two normal forms $\neg p \vee (\neg q \wedge \neg r)$ and $(\neg p \wedge \neg p) \vee (\neg p \wedge \neg r) \vee (\neg q \wedge \neg p) \vee (\neg q \wedge \neg r)$ according to whether rule (R3) or (R6) is applied first. The exact outcome of the normal form depends on the strategy of the narrowing system, which determines the order of sub-formula selection and rule application. The implemented narrowing system considers first the sub-formulas in *prefix traversal* order first and then the rules in the *input order.* Using this strategy the generated normal form of the above example will be $\neg p \vee (\neg q \wedge \neg r)$ which is intuitively less complex to evaluate than the other normal form.

When one learns about disjunctive normal forms in mathematical logic one is told to first apply rules (R3) to (R5) in order to move the negation inside and then rules (R6) to (R7) in order to move the disjunction outside. Unfortunately, when rule (R8) is included such a partition of the rules is no longer possible. Take for example the formula $\neg\exists x(p \wedge (q \vee r))$. To get the normal form $\neg\exists x(p \wedge q) \wedge \neg\exists x(p \wedge r)$ the rules (R6), (R8) and (R4) have to be applied in that order. Hence the rules (R3) to (R8) cannot be applied sequentially and in isolation.

Figure 3.8 shows the main part of the narrowing system and a fragment of the rewriting rules for ProQuel. The application of the predicate `norm/2` to $\neg(p \wedge (q \vee r))$ is also shown. The rewriting rules are represented by facts for the infix operator `==>`. The narrowing system uses the predicates `get` and `put` to traverse and modify the terms. `get(F,N,S)` is intended to return the `N`-th direct sub-formula of `F` in `S`, whereas `put(F,N,S,G)` is intended to construct in `G` the formula that is obtained from `F` by replacing the `N`-th direct sub-formula by `S`.

The narrowing system implemented so far is not very fast. The reason for that is

```
%% +term ==> -term
imp(X,Y) ==> or(not(X),Y).
all(V,X) ==> not(ex(V,not(X))).
not(and(X,Y)) ==> or(not(X),not(Y)).
...

%% try1(+term,-path,-term)
try1(X,[],Y) :- X==>Y.
try1(X,[N|P],Y) :- get(X,N,Z), try1(Z,P,T), put(X,N,T,Y).

%% try2(+term,+path,-path,-term)
try2(X,[],P,Y) :- try1(X,P,Y).
try2(X,[_|_],[],Y) :- X==>Y.
try2(X,[N|P],[N|Q],Y) :- get(X,N,Z), try2(Z,P,Q,T), put(X,N,T,Y).
try2(X,[N|_],[M|Q],Y) :- get(X,M,Z), M>N, try1(Z,Q,T), put(X,M,T,Y).

%% norm(+term,+path,-term)
norm(X,P,Y) :- try2(X,P,Q,Z), !, norm(Z,Q,Y).
norm(X,_,X).

?- norm(not(and(p,or(q,r))),[],X)
X=or(not(p),and(not(q),not(r)))
```

Figure 3.9: Code Fragment Improved Normalization

depth	1	2	3	4	5	6
naive	0.02	0.04	0.08	1.40	32.12	97.30
improved	0.02	0.08	0.08	0.52	2.68	5.58

Table 3.2: Normalization Benchmarks

the fact that the whole formula is traversed on every normalization step. This can be avoided. Obviously the sub-formulas on the left hand side of the last modified sub-formula are not affected and therefore they must not be checked again. Hence it is possible to restrict the search space considerably without affecting the rule application strategy. Figure 3.9 shows the improved version of the narrowing system. The predicates now have an additional argument for the path leading to the last modified sub-formula. The path is simply a list of direct sub-formula indices. The predicate try1 performs the search above the last modified sub-formula, whereas the predicate try2 performs the search on the right hand side of the last modified sub-formula. The left hand side of the last modified sub-formula is no longer searched.

The better performance of the second version was verified by some benchmarks. The timings in table 3.2 show that the overhead to handle the paths in the second version can be neglected. They also indicate that the asymptotic behaviour of the two versions is different. The timings include the runtime of the two versions on 10 different, randomly generated, balanced formulas for every depth. The timings

```
%% varof(+term,-var).
varof(and(A,B),X) :- varof(A,X); varof(B,X).
varof(not(A),X) :- varof(A,X).
varof(ex(V,A),X) :- varof(A,X), X\==V.
...

%% out(+term,+varlist)
out(X,S) :- X=var(V); \+ (varof(X,V), \+member(V,S)).

%% in(+term,+varlist)
in(X,S) :- \+ (varof(X,V), \+member(V,S)).

%% ok(+term,+varlist)
ok(ls(E,F),S) :- in(E,S), in(F,S).
ok(eq(E,F),S) :- (in(E,S), out(F,S)); (out(E,S), in(F,S)).
ok(and(A,B),S) :- ok(A,S), setof(V,(varof(A,V);member(V,S)),R), ok(B,R).
...
```

Figure 3.10: Code Fragment Allowedness Check

are show in seconds and they were obtained on a SUN 3/480 with the the SICStus Prolog compiler.

3.6 Allowedness Check

Recall that the aim of ProQuel is to calculate the answer set of a query. In general such an answer set may turn out to be infinite. Consider for example the following query:

$$\neg p(X)$$

Assume that the argument of p ranges over the natural numbers and that the query $p(X)$ gives the answer $\{X = 0\}$. Then clearly $\neg p(X)$ must give the infinite answer $\{X = 1, X = 2, X = 3, \ldots\}$. Some programming languages may handle infinite sets to some extent. For example Prolog, because of its tuple oriented processing, may return an infinite answer set by backtracking. Infinite sets are not welcome in commercial databases because a set oriented processing is desired. Therefore we have restricted ourselves to *allowed* queries and rules as defined in section 5.4.

In LogiQuel [MWW89] and NewQuel [Bur89] a syntactical criterion was used that was essentially derived from allowedness in [TS88]. The criterion was checked during the translation phase and it was therefore highly target dependent. In ProQuel [Hel91] the check was successfully isolated from the target. The ProQuel allowedness check is now part of the core and the class of accepted queries and rules is always the same independent of the actual target.

Figure 3.10 shows a code fragment of the current allowedness check. The program

```
notstrat(P,P,0,L).
notstrat(P,Q,S,L) :-
   \+member(Q,L), db_fetch(deps(Q,_,Q1,S2)),
   S1 is S*S2, notstrat(P,Q1,S1,[U|L]).

notstrat(P,Q,S) :- notstrat(P,Q,S,[P]).
```

Figure 3.11: Code Fragment Stratification Check

is a direct translation of definitions 5.55 and 5.56 in section 5.4. Note that lists are used to represent sets of variables. A minimum of two predicates, namely `member` and `setof`, are used to manipulate these sets.

3.7 Stratification Check

The standard model as defined in section 5.3 is defined only for stratified rules. Therefore provisions have to be made that the user is not able to enter rules that are not stratified.

In LogiQuel [MWW89] and NewQuel [Bur89] the stratification was checked during the plan generation. As a result it was possible that the knowledge base was not fully stratified during the query evaluation. More disturbingly, the source of a stratification violation could not be detected during its creation. Therefore in ProQuel the stratification is checked in advance whenever a rule is asserted.

To check the whole knowledge base for stratification after each assertion of a rule would be very inefficient. In ProQuel an incremental algorithm is used. When new rule for p is accepted, a new set of signed dependencies $p \rightarrow^{m_i} q_i$ with $m_i \in \{+,-\}$ will have to be added to the dependency graph. Clearly a new cycle would go through p and it suffices only to look for such cycles that use exactly one of the new dependency $p \rightarrow^{m_i} q_i$. Hence the new dependencies can be checked independently for a new induced negative cycle.

The corresponding Prolog program is given in figure 3.11. A dependency $p \rightarrow^{m} q$ is stored as a $deps(p, k, q, s)$ fact where k is a unique id of the rule to which the dependency belongs and where $s = 0$ and $s = 1$ stand for $m = -$ and $m = +$ respectively. The facts are stored in the target and are accessed via the meta database interface predicate `db_fetch` (see section 4.1). The predicate $notstrat(p, q_i, s_i)$ must fail for all the new dependencies $p \rightarrow^{m_i} q_i$ before the rule and its new dependencies can be entered into the meta database. The predicate *notstrat* simply traverses the dependency graph until it finds a negative cycle. A list with the visited nodes accompanies the predicate to avoid infinite looping.

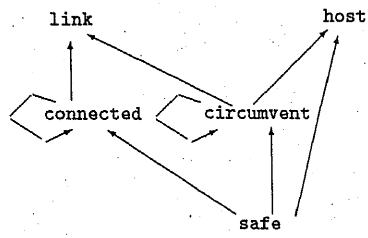

Figure 3.12: Dependency Graph

3.8 Plan Generation

When a query is posed ProQuel will first generate a plan. The plan is used later on in the materialization of the tables. It contains the order of evaluation of the required tables and information that indicates whether a recursive predicate has to be evaluated or not.

The plan generation is based on the identification of the *strongly connected components* (s.c.c.) in the dependency graph. The s.c.c. problem is well understood and there are linear (in the number of edges and vertices) algorithms to compute them [Mel84, p.21]. In LogiQuel [MWW89] an algorithm from the literature was adapted [Taj72]. In NewQuel [Bur89] a slight modification was made such that the plan was generated in one pass. The modified algorithm was successfully ported from Modula-2 to Prolog for the ProQuel system.

Figure 3.12 shows the unsigned dependencies of the example in section 2.5. In general a plan consists of a list of elements of the form $P - S$. An element $P - []$ indicates a non-recursive predicate P. An element $P - S$ with S non-empty indicates a recursive predicates P that belongs to the s.c.c. S. The modified algorithm yields the following plan for the query $safe(1,4)$:

1. link-[]
2. connected-[connected]
3. host-[]
4. circumvent-[circumvent]
5. unsafe-[]

Figure 3.13 shows the Prolog predicates `sccforpred` and `sccforlist` that are used to compute the plan. The predicate `sccforpred` has a predicate name `Pred` as input in the first argument, whereas the predicate `sccforlist` has a list of predicate names `List` instead. Both predicates additionally have a `Path` and a `Plan1` as input arguments and a `Plan2` and a `Scc` as output arguments. `Path` holds the current path through the dependency graph. `Plan1` and `Plan2` are used to accumulate the plan. `Scc` returns the actual s.c.c. The first rule of `sccforpred` detects a recursive predicate. The second rule for `sccforpred` detects a predicate name that

```
%% sccforpred(+pred,+stack,+plan,-plan,-scc)
sccforpred(Pred,Path,Plan,Plan,[Pred]) :-
  member(Pred,Path), !.
sccforpred(Pred,_,Plan,Plan,[]) :-
  member(Pred1-Scc1,Plan), (Pred1=Pred; member(Pred,Scc1)), !.
sccforpred(Pred,Path,Plan1,Plan2,Scc) :-
  setof(Pred1,db_fetch(deps(Pred,_,Pred1,_),List),
  sccforlist(List,[Pred|Path],Plan1,Plan3,Scc1),
  (member(Pred2,Scc1), member(Pred2,Path) ->
    Plan2=Plan3, Scc=[Pred|Scc1];
    Plan2=[Pred-Scc1|Plan3], Scc=[]).

%% sccforlist(+predlist,+stack,+plan,-plan,-scc)
sccforlist([],_,Plan,Plan,[]).
sccforlist([Pred|List],Path,Plan1,Plan2,Scc) :-
  sccforpred(Pred,Path,Plan1,Plan3,Scc1),
  sccforpred(List,Path,Plan3,Plan2,Scc2),
  setof(Pred1,(member(Pred1,Scc1);member(Pred1,Scc2)),Scc).
```

Figure 3.13: Code Fragment Plan Generation

has already been treated. The last rule for `sccforpred` first calls `sccforlist`. It then will append the current predicate name to the current s.c.c., in case that the s.c.c. is not completed. Otherwise a new entry into the current plan is generated. `sccforlist` simply applies `sccforpred` on all elements of the list and collects the generated s.c.c.s.

3.9 Materialization

The materialization of rules is managed by the core which is able to map these duties to the elementary operations of the target. The ProQuel system always behaves the same independent of the underlying target because the materialization is managed by the core. For the sake of simplicity we have omitted various optimizations that have been pursued in LogiQuel [MWW89] and NewQuel [Bur89]. ProQuel does not apply a semi-naive strategy to linear rules and it does not generate views for non-recursive predicates.

The materialization needs a cache table that consists of the names of the currently materialized tables. The cache table has the invariant that if it contains the table p and p depends on q, then it also contains q. Hence a cone of tables is always materialized. The cache table is used to avoid duplicate work. Every table that is used directly or indirectly in a query is only materialized once. The cache table and the corresponding materializations have to be partially flushed in the case that rules or facts are modified.

The materialization of rules is done according to the presence of recursion. For

```
%% execscc(+planelem)
execscc(P-_) :-
  db_fetch(cache(P)), !.
execscc(P-[]) :- !,
  (db_fetch(rule(P,_,R)), insertrule(R), fail; true),
  db_insert(cache(P)).
execscc(_-S) :-
  (member(Q,S),
     db_fetch(rule(Q,KEY,R)),
     \+ (db_fetch(deps(_,KEY,U,_)), member(U,S)),
     insertrule(R), fail; true),
  findall(C,(member(Q,S),ev_count_tuples(Q,C)),L),
  iterfixpoint(L,S),
  (member(Q,S), db_insert(cache(Q)), fail; true).

%% iterfixpoint(+countlist,+scc)
iterfixpoint(L,S) :-
  (member(Q,S),
    db_fetch(rule(Q,KEY,R)),
    \+ \+ (db_fetch(deps(_,KEY,U,_)), member(U,S)),
    insertrule(R), fail; true),
  findall(C,(member(Q,S),ev_count_tuples(Q,C)),L1),
  (L1\==L -> iterfixpoint(L1,S); true).
```

Figure 3.14: Code Fragment Materialization

non-recursive predicates, every rule is simply materialized. Recursive predicates are materialized by a naive fix-point iteration. In this case, all non-recursive rules are first materialized. Then the materialization of the recursive rules is iterated as long as there are changes. The materialization of a recursive predicate is always done for the whole s.c.c. of the predicate.

Figure 3.14 shows a code fragment of the materialization. The predicate `execscc` is responsible for the materialization of an element of the plan. In the first rule the predicate `execscc` simply does nothing in the case of a predicate that is already in the cache table. The second rule of `execscc` handles non-recursive predicates. In the last rule of `execscc` recursive predicates are handle. The non-recursive rules are materialized and `iterfixpoint` is called with the the s.c.c. and the counts of the tables. The predicate `iterfixpoint` then materializes the recursive rules until all the counts of the tables are stable. The predicate `insertrule` is intended to materialize one rule.

Chapter 4

ProQuel Targets

Figure 4.1 shows the overall architecture of ProQuel. The ProQuel system features the support of different target databases. The target databases are responsible for the storage of the data and the evaluation of the queries. The current ProQuel implementation includes a SQL target, a POSTGRES target and a Prolog target. The ProQuel system cannot be regarded as a heterogeneous database system as the target databases can only be used separately and in isolation.

In the following a more detailed description of the ProQuel targets is given. The emphasis is on the translation between the ProQuel language and the corresponding target language. The chapter starts with a definition of the target interface in section 4.1. It then continues with the SQL target, the POSTQUEL target and Prolog target in section 4.2, 4.3 and 4.4 respectively.

The three targets demonstrate the usefulness of our notion of allowedness. They show that it is possible to translate allowed formulas that even contain quantifiers and negation to three totally different targets. In particular the SQL and POSTQUEL targets show that it is also possible to have a purely declarative and uniform access to different relational database systems in the form of a deductive database interface.

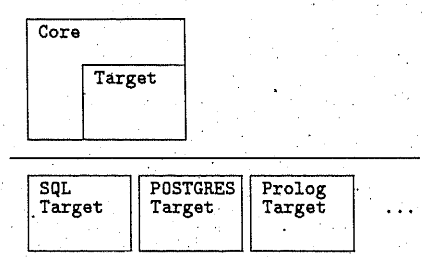

Figure 4.1: Overall ProQuel Architecture

`db_insert(+Fact)`:	Insert the fact.
`db_fetch(-Fact)`:	Successively retrieve the matching facts.
`db_delete(-Fact)`:	Remove all the matching facts at once.

Figure 4.2: Meta Database Interface

pred(Name:str,Definition:str)
rule(Name:str,Key:int,Rule:str)
deps(Fromname:str,Key:int,Toname:str,Sign:int)
cache(Name:str)

Figure 4.3: Meta Database Schema

4.1 Target Interface

The target interface consists of a set of Prolog predicates that have to be supported by each target. The interface itself splits into the *meta database* and the *evaluator*. The meta database is responsible for the meta data of the core. It is retrieved and manipulated through the meta database interface. The evaluator is used by the core to perform the query evaluation. The core must utilize the minimal functionality of the evaluator in order to materialize the right tables.

The meta database must provide relational data with attributes that range over the atomic domains of Prolog. A tuple $(c_1, \ldots, c_n)$ that belongs to a table r is written as a Prolog fact in the form $r(c_1, \ldots, c_n)$. The meta database must be able to retrieve and manipulate such facts. The target database is free in the form of storage of these facts. The interface to the meta database consists of the `db_` predicates that are given in figure 4.2.

Among the relations that have to be supported by the meta database are the relation `pred` which is used to store the predicate definitions, the relation `rule` which contains the rules, the relation `deps` which is used for the dependency graph and the relation `cache` which indicates tables that are already materialized. The meta data contains structures like rules and definitions that are all stored as strings and converted from and to Prolog by DCG parsers and unparsers in need. The meta database schema is given in figure 4.3.

The evaluator is controlled by `ev_` predicates which are given in figure 4.4. There are predicates to disconnect and connect the database (`ev_open_db` and `ev_close_db`), predicates to create, clear and destroy tables (`ev_create_tab`, `ev_clear_tab` and `ev_drop_tab`) and predicates to evaluate queries (`ev_count_tuples`, `ev_insert_res` and `ev_query_res`). The predicates `ev_insert_res` and `ev_query_res` both have an `Exprs` and a `Cond` argument. These arguments denote a set `{Exprs | Cond}` which is inserted in the specified table in the case of `ev_insert_res` and which is returned by backtracking in the case of `ev_query_res`. The condition `Cond` is given as a term that represents a normalized formula and the expression list `Exprs` is given

`ev_open_db(+Name):`	Connect with database.
`ev_close_db:`	Disconnect from database.
`ev_create_tab(+Name,+Sorts):`	Create a table.
`ev_clear_tab(+Name):`	Remove all tuples from a table.
`ev_drop_tab(+Name):`	Destroy a table.
`ev_count_tuples(+Name,-Count):`	Count tuples in a table.
`ev_insert_res(+Name,+Exprs,+Cond):`	Insert tuples into a table.
`ev_query_res(+Exprs,+Cond,-Tuple):`	Retrieve tuples.

Figure 4.4: Evaluator Interface

```
stat    = create(str,*sort) + drop(str) + insert(str,selstat) + ...

selstat = sel(*expr,*from,*where) + union(selstat,selstat) + ...

from    = alias(str,int).

where   = notexists(selstat) + eq(expr,expr) + ls(expr,expr) + ...

expr    = attr(int,int) + cint(int) + add(expr,expr) + ...
```

Figure 4.5: Query Tree Structure

as a list of terms that represent expressions. The central theme of this chapter is the translation of `{Exprs | Cond}` into an appropriate notion of the target language.

4.2 The SQL Target

The SQL language was the first target language that was supported in the LogiQuel project. The succession of deductive database systems described in [Bur89], [Wüt91] and [Hel91] were exclusively designed for SQL. SQL appears to be widely available on different database systems and the database system of the LogiQuel project was Oracle/SQL. The experience with Oracle/SQL was satisfying because it is a robust database system with an acceptable performance.

In [Bur89], [Wüt91] and [Hel91] the SQL queries were directly generated by using string concatenation. In ProQuel another approach has been taken: Instead of a query string, the translator generates a query tree. The query tree is then used in a DCG unparser to generate the final string. This approach has many advantages as the program becomes more readable and the expensive string concatenation is avoided. Figure 4.5 shows a sketch of the Prolog structure that is used to represent the target query tree.

The translation from formulas to query trees is done left to right. For every new occurrence of a predicate a new *alias* is generated. In this way name clashes between predicates that occur more than once are avoided. The translation also needs a

```
%% trans_form(+term,+rec,-rec)
trans_form(and(E,F),I,O) :-
  trans_form(E,I,H), trans_form(F,H,O).
trans_form(not(E),rec(L,F,W,A),rec(L,F,[notexists(sel([],F1,W1))|W],B)) :-
  trans_form(E,rec(L,[],[],A),rec(_,F1,W1,B)).
trans_form(ex(V,E),rec(L,F,W,A),rec(L3,F1,W1,A1)) :-
  get_attr(L,V,H), put_attr(L,V,undef,L1),
  trans_form(E,rec(L1,F,W,A),rec(L2,F1,W1,A1)),
  put_attr(L2,V,H,L3).
trans_form(pred(P,E),rec(L,F,W,A),rec(L1,[alias(P,A)|F],W1,A1)) :-
  A1 is A+1,
  trans_where(E,A,0,L,L1,W,W1).
trans_form(eq(P,Q),rec(L,F,W,A),rec(L1,F,W1,A)) :-
  (P=var(V), get_attr(L,V,undef) ->
     trans_expr(Q,L,X), put_attr(L,V,X,L1), W1=W;
   Q=var(V), get_attr(L,V,undef) ->
     trans_expr(P,L,X), put_attr(L,V,X,L1), W1=W;
   trans_expr(P,L,K), trans_expr(Q,L,J), L1=L, W1=[eq(K,J)|W]).
trans_form(ls(P,Q),rec(L,F,W,A),rec(F,[ls(K,J)|W],A)) :-
  trans_expr(P,L,K), trans_expr(Q,L,J).
...
```

Figure 4.6: Code Fragment SQL Translation

symbol table which is accessed trough `get_attr` and `put_attr` predicates. The symbol table records the association between variables of the formula and expressions of the query tree. If a variable occurs for the first time in a predicate then an alias number and column number pair is generated as a coordinate for the variable. A variable may also occur on one side of an assignment for the first time. The other side of the assignment is then associated with the variable. The negation of a formula is translated to a NOT EXISTS sub-query. The translation is sound for allowed formulas.

Figure 4.2 shows a code fragment of the translation process. The Prolog predicate `trans_form` is used to translate the or-free parts of a normal form. It appropriately handles the different cases of the allowed formula and carries along an environment record `rec(L,F,W,A)` which contains the current symbol table `L`, the current from-part `F`, the current where-part `W` and the alias counter `A`. The from-part `F` is a list of `alias(Name,Alias)` pairs and the where-part `W` is a list of where conditions. The first rule simply translates a conjunctions by carrying along the environment record. The second rule generates a NOT EXISTS condition for the negation of a formula. In the next rule, to reflect the scoping of quantifiers, the quantified variable is first hidden from the symbol table, the quantified formula is then translated and the variable is finally entered again into the symbol table. The following rule translates a predicate occurrence by calling `trans_where` which generates the appropriate where conditions and symbol table entries. Finally, the last two rules show the translation of the built-ins = and <. To illustrate the translation consider the following query:

$$p(X,1) \wedge \neg \exists Y(q(X,Y) \wedge Y > 2)$$

The query is already in normal form and consists of only one or-free part. The translation will be given for the output variable `X`.. The translation consists of the Prolog terms `Expr1`, `From1` and `Where1` for the sub-formula $q(X,Y) \wedge Y > 2$ and of the Prolog terms `Expr2`, `From2` and `Where2` for the main formula.

```
expr1    = [attr(0,0),attr(1,1),attr(0,1)]
from1    = [alias('q',1)]
where1   = [eq(attr(0,0),attr(1,0)),gr(attr(1,1),cint(2))]

expr     = [attr(0,0)]
from     = [alias('p',0)]
where    = [eq(attr(0,1),cint(1)),notexists(sel(Expr1,From1,Where1))]
```

The query tree of the main formula is `sel(Expr2,From2,Where2)`. If the DCG unparser is applied to the query tree the following final SQL string will be obtained:

```
SELECT A0.C0, A0.C1
FROM p A0
WHERE A0.C1=1
AND NOT EXISTS (
  SELECT A0.C0, A1.C1
  FROM q A1
  WHERE A0.C0=A1.C0
  AND A1.C1>2)
```

4.3 The POSTQUEL Target

The deductive database systems described in [Kel91] and [Ber91] were both to run on top of the experimental POSTGRES system, a successor to INGRES, developed at the University of Berkeley. It was hoped that several of its features (in particular the abstract data type facility) could be used to our advantage. POSTGRES supports the language POSTQUEL, a successor of the QUEL language.

In [Kel91] and [Ber91] the POSTQUEL string was directly generated using string concatenation. As in the SQL target, the POSTQUEL target first generates a query tree instead. The query tree is a Prolog term and its data structure does not differ considerably from the one that is used for the SQL target. The translation to POSTQUEL is very similar to the SQL translation. Aliases are generated, a symbol table is needed and the first occurrence of a variable is treated in the same way. However the two targets differ considerably in the treatment of the negation. In the current POSTGRES version sub-queries were not available. Therefore the negation was implemented by means of temporary relations and the DELETE statement. The DELETE statement has the following form:

```
%% trans_form(+term,+rec,-rec)
trans_form(and(E,F),I,O) :-
  trans_form(E,I,H), trans_form(F,H,O).
trans_form(not(E),rec(L,F,W,A,D),rec(L1,[alias(N,A)],[],A2,D2)) :-
  concat('_',A,N),
  trans_assoc(L,A,O,S,L1),
  pg_exec2(into(N,S,F,W)),
  A1 is A+1,
  trans_form(E,rec(L1,[alias(N,A)],[],A1,[N|D]),rec(_,F1,W1,A2,D2)),
  pg_exec2(delete(N,F1,W1)).
trans_form(ex(V,E),rec(L,F,W,A,D),rec(L3,F1,W1,A1,D1)) :-
  get_attr(L,V,H), put_attr(L,V,undef,L1),
  trans_form(E,rec(L1,F,W,A,D),rec(L2,F1,W1,A1,D1)),
  put_attr(L2,V,H,L3).
trans_form(pred(P,E),rec(L,F,W,A),rec(L1,[alias(P,A)|F],W1,A1)) :-
  A1 is A+1,
  trans_where(E,A,O,L,L1,W,W1).
trans_form(eq(P,Q),rec(L,F,W,A,D),rec(L1,F,W1,A,D)) :-
  (P=var(V), get_attr(L,V,undef) ->
     trans_expr(Q,L,X), put_attr(L,V,X,L1), W1=W;
   Q=var(V), get_attr(L,V,undef) ->
     trans_expr(P,L,X), put_attr(L,V,X,L1), W1=W;
   trans_expr(P,L,K), trans_expr(Q,L,J), L1=L, W1=[eq(K,J)|W]).
trans_form(ls(P,Q),rec(L,F,W,A,D),rec(F,[ls(K,J)|W],A,D)) :-
  trans_expr(P,L,K), trans_expr(Q,L,J).
...
```

Figure 4.7: Code Fragment POSTQUEL Translation

```
DELETE r
FROM r1, ..., rn
WHERE c
```

The logical essence of the DELETE statement is as follows. Let `r'` denote the content of the table `r` after the execution of the DELETE statement. Vaguely formulated the following condition holds:

$$r' \leftrightarrow r \wedge \neg(r \wedge r_1 \wedge \ldots \wedge r_n \wedge c)$$

Now consider the translation of a negation $A \wedge \neg B$. It is easy to verify that $A \wedge \neg B \equiv A \wedge \neg(A \wedge B)$ holds for arbitrary formulas. Hence it is possible to translate a negation $A \wedge \neg B$ by first materializing A into a relation r and then translating the negative sub-formula B into a query with from-part $r_1, \ldots, r_n$ and where-part c. The result of $A \wedge \neg B$ may then be obtained by issuing a DELETE statement of the previously mentioned form.

Because of the DELETE statement, the translation to POSTQUEL is more difficult to implement than the translation to SQL. Figure 4.3 shows a code fragment of the translation to POSTQUEL. This time the predicate trans_form carries along a data structure rec(L,F,W,A,D) which again contains the current symbol table L, the current from-part F, the current where-part W and the alias counter A. There is a new list D with the names of the temporary tables that are generated for negative sub-formulas and that have to be destroyed after the processing of the main formula. To illustrate the translation consider again the following query:

$$p(X,1) \wedge \neg\exists Y(q(X,Y) \wedge Y > 2)$$

Assume that the left part of the negation, i.e. the sub-formula $p(X,1)$, is materialized in the table h. The following sequence is generated in the translation of the main formula to POSTQUEL:

```
RETRIEVE INTO h (a0.c0) /* materialize p(X,1) in temp. rel. */
FROM a0 IN p
WHERE a0.c1=1;

DELETE a0               /* delete q(X,Y)&Y>2 from temp. rel. */
FROM a0 IN h, a1 IN q
WHERE a0.c0=a1.c0
AND a1.c1>2;

RETRIEVE (a0.c0)        /* main result in temp. rel. */
FROM a0 IN h;

DESTROY h;              /* destroy temp. rel. */
```

4.4 The Prolog Target

The Prolog target was mainly issued to gain experience with a target that works in main memory. The Prolog target was implemented on the same Prolog system as the core, namely SICStus Prolog. The experience with SICStus as an implementation language for the target was very satisfying as it was with the core.

Not surprisingly, the Prolog target is the simplest one. It is not necessary to generate a query string because it is possible directly to generate a query tree that can be passed to Prolog for evaluation. There is also no need to generate aliases because Prolog doesn't know attribute names. As a query language, Prolog is a position oriented language like the ProQuel language itself. A symbol table is still needed to store the association between variables and Prolog expressions. The translation to Prolog is almost one to one because most of the logical connectives and built-ins have a direct correspondence in Prolog. To translate the ProQuel negation the built-in operator \+ is used which implements negation as failure.

In figure 4.4 a code fragment of the Prolog translation is given. The only structure that is carried along is the symbol table L. The predicate trans_form directly generates a new Prolog query in its last argument. The translation is merely a

```
%% trans_form(+term,+symtab,-symtab,-prologquery)
trans_form(and(A,B),L,R,Q) :-
  trans_form(A,L,H,Q1), trans_form(B,H,R,Q2), make_and(Q1,Q2,Q).
trans_form(not(A),L,L,\+Q) :-
  trans_form(A,L,_,Q).
trans_form(ex(V,E),L,L3,Q) :-
  get_attr(L,V,H), put_attr(L,V,undef,L1),
  trans_form(E,L1,L2,Q),
  put_attr(L2,V,H,L3).
trans_form(pred(P,A),L,R,Q) :-
  trans_where(A,L,R,E,Q1), Q2=..[P|E], make_and(Q1,Q2,Q).
trans_form(eq(E,F),L,R,Q) :-
  (E=var(V), get_attr(L,V,undef) ->
     trans_expr(F,L,X,Q), put_attr(L,V,X,R);
   F=var(V), get_attr(L,V,undef) ->
     trans_expr(E,L,X,Q), put_attr(L,V,X,R);
   trans_expr(E,L,X,Q1), trans_expr(F,L,Y,Q2),
   make_and(Q1,Q2,Q3), make_and(Q3,X@<Y,Q), R=L).
trans_form(ls(E,F),L,L,Q) :-
  trans_expr(E,L,X,Q1), trans_expr(F,L,Y,Q2),
  make_and(Q1,Q2,Q3), make_and(Q3,X@<Y,Q).
...
```

Figure 4.8: Code Fragment Prolog Translation

replacement of the ProQuel operators by the appropriate Prolog operators. To illustrate the translation consider again the following query:

$$p(X,1) \wedge \neg\exists Y(q(X,Y) \wedge Y > 2)$$

The translation to Prolog is as follows:

```
p(X,1), \+ (q(X,Y), Y>2)
```

Prolog has an ability that is not found in the SQL or POSTQUEL language. Prolog is able to handle *temporary variables* which means that we can freely introduce and use a variable in a query. This is not the case for the SQL and POSTQUEL language. Consider for example the following query:

$$p(X,Y) \wedge H = 100 * X \wedge H - 10 < Y \wedge H + 10 > Y$$

The translation to Prolog is as follows:

```
p(X,Y), H is 100*X, H1 is H-10, H1<Y, H2 is H+10, H2>Y
```

The translation to SQL is as follows:

```
SELECT A0.C0, A0.C1, A0.C0*100
FROM p A0
WHERE A0.C0*100-10<A0.C1
AND A0.C0*100+10>A0.C1
```

In the translation to Prolog the expression $100*X$ is evaluated only once and associated with the variable. In the translation to SQL the expressions $100*X$ has to be substituted because it is not possible to introduce a variable in an SQL query. As a consequence in general SQL expressions become bigger and it is hoped that SQL optimizers do have common subexpression elimination in order to reduce this redundancy.

Chapter 5

ProQuel Semantics

This chapter motivates the use of the standard model semantic for stratified knowledge bases. ProQuel is only one application of the framework presented here. The framework consists of many sorted structures and languages presented in 5.1. Section 5.2 contains some results concerning rules and their completion. In section 5.3 the standard model for stratified knowledge bases is presented. Finally the computational tractability of the standard model is considered in section 5.4. The main result of this chapter is a semantic for a class of knowledge bases with arbitrary domains, functions and built-ins.

5.1 Structure and Language

The model M_D assigned to a knowledge base D will be a many sorted first order structure. A many sorted structure M has a set s^M for every sort symbol s. The reason for using many sorted structures is that sorts permit a natural way of expressing the domain concept of relational databases. The use of sorts in deductive databases is not new and has already been proposed in the proof theoretic setting of [Llo85], [Llo87, p.143]. Sorts have also proven good in mechanical theorem proving [Wal84]. Instead of the term sort one can also use the equivalent terms type, domain or universe.

In defining structures one first has to define the notion of a signature. A signature defines the kind of sorts, functions and relations that are used by a structure. There are many different structures that can belong to the same signature. The many sorted signatures and structures are a straight forward generalizations of the unsorted signatures and structures used in first order model theory [Wan52]:

Definition 5.1: *A* signature Σ
consists of the following ingredients:

i) A set S of sort symbols.

ii) A set F of function symbols *and for every $f \in F$ a*
function signature $\sigma(f) = s_1 \times \ldots \times s_n \to s$ *with* $s_1, \ldots, s_n, s \in S$.

iii) A set R of relation symbols *and for every $r \in R$ a* relation signature $\rho(r) = s_1 \times \ldots \times s_n$ *with* $s_1, \ldots, s_n \in S$.

Definition 5.2: *A* structure M *of a signature* Σ *consists of the following ingredients:*

i) *For every* $s \in S$ *a* set s^M, $s^M \neq \emptyset$.

ii) *For every* $f \in F$ *with* $\sigma(f) = s_1 \times \ldots \times s_n \rightarrow s$ *a* function $f^M : s_1^M \times \ldots \times s_n^M \rightarrow s^M$.

iii) *For every* $r \in R$ *with* $\rho(r) = s_1 \times \ldots \times s_n$ *a* relation $r^M \subseteq s_1^M \times \ldots \times s_n^M$.

The model M will contain predefined domains, functions and relations. It is not considered that the user can define new domains and functions. The user is only permitted to define new relations. The predefined relations are called built-ins, the user definable ones predicates. For this purpose the notion of a signature and structure is extended a little bit. Clearly there can be arbitrary functions and built-ins. Nevertheless the following definition prescribes that there is at least one built-in $=_s$ for every sort s such that elements of the predefined domains can be tested for equivalence.

Definition 5.3: *The relation symbols* R *of a signature* Σ *are partitioned into the* built-in symbols B *and the* predicate symbols P. *For every sort* $s \in S$ *there has to be an* equivalence symbol $=_s \in B$ *with* $\rho(=_s) = s \times s$. *In a structure* M *of signature* Σ *the equivalence symbol has to be interpreted by the standard* $=_s^M = \{(m, m) \mid m \in s^M\}$.

If data structures are understood as data types equipped with constructors, destructors and operations among them, then most of them are many sorted first order structures. As long as functions do not become first class objects it seems that many sorted first order structures are conceptually sufficient to model data structures. Some examples:

Example 5.4: *The* Herbrand universe *is an example of a single sorted first order structure. It plays an important role in logic programming. In a Herbrand universe the functions do not evaluate. This can be seen in Prolog which is the most prominent member of the logic programming language family. In Prolog the term* $1+2$ *does not automatically evaluate to* 3, *instead it evaluates to itself.*

Example 5.5: *The* NF2 data model *can be regarded a many sorted first order structure. In the NF2 data model the attributes are allowed to be tables again. Hence if* $s_1, \ldots, s_n$ *are sorts then* $tab(s_1, \ldots, s_n)$ *is again a sort. Suppose that tables are finite sets then we can define the NF2 data model to contain the many sorted structures* M *with* $tab(s_1, \ldots, s_n)^M = \{T \subseteq s_1^M \times \ldots \times s_n^M \mid T \textit{ finite}\}$. *The model also contains some table specific built-ins like unnest and nest.*

Example 5.6: *The* ProQuel data model *is a many sorted first order structure too. The sorts are* int, $float$ *and* str. *Built-ins like* $=$, $<$. $\ldots$ *and functions like* $+$, $-$, $\ldots$ *are also contained in the model.*

To talk about many sorted structures of a given signature Σ a many sorted language is used. The language is based on an alphabet that contains the sort symbols S, the function symbols F and the relations symbols R. The alphabet also contains infinite countable many variables V_s for every $s \in S$ and some special symbols. The sets S, F, R and V_s for $s \in S$ are all considered to be pairwise disjoint. The language is made up of sorted terms and formulas.

Definition 5.7: *$Term_s$ denotes the set of* terms *of sort s.*
It is inductively defined as follows:

i) If $x \in V_s$ then $x \in Term_s$.

ii) If $t_1 \in Term_{s_1}, \ldots, t_n \in Term_{s_n}$, $f \in F$ and $\sigma(f) = s_1 \times \ldots \times s_n \to s$ then $f(t_1, \ldots, t_n) \in Term_s$.

Definition 5.8: *$Form$ denotes the set of* formulas.
It is inductively defined as follows:

i) $\bot, \top \in Form$.

ii) If $t_1 \in Term_{s_1}, \ldots, t_n \in Term_{s_n}$, $r \in R$ and $\rho(r) = s_1 \times \ldots \times s_n$ then $r(t_1, \ldots, t_n) \in Form$.

iii) If $A \in Form$ then $\neg A \in Form$.

iv) If $A, B \in Form$ then $(A \wedge B), (A \vee B), (A \to B), (A \leftrightarrow B) \in Form$.

v) If $x \in V_s$ and $A \in Form$ then $\exists x A, \forall x A \in Form$.

The symbols $\bot$ and $\top$ are used for `false` and `true` respectively. Constants are regarded as function symbols with zero arity. Formulas, that are exactly formed according to the above definition, become rather clumsy. Therefore relations like $=$, $<, \ldots$ and functions like $+, -, \ldots$ are usually written infix, i.e. $(A = B) := =(A, B)$ and $(A + B) := +(A, B)$. The formulas then tend to have a lot of parenthesis, i.e. $(employee(X) \wedge (Y = (X + 2)))$. It is then necessary to introduce some standard operator precedences that allow the omission of parenthesis, i.e. $employee(X) \wedge Y = X + 2$. These conventions make it more comfortable to write formulas and they should only be regarded as shorthand notations for the original formulas.

In the following the value of a term and the truth value of a formula are defined in detail. The value and the truth value depend on a structure M and on a assignment θ. An assignment θ for M is a mapping from sorted variables to domain elements such that $\theta(x) \in s^M$ for $x \in V_s$. $\theta\frac{m}{x}$ denotes the assignment that is obtained from θ by replacing the value of $x \in V_s$ by the value $m \in s^M$. The truth value is only given for the fragment $\top, \neg, \wedge$ and $\exists$. The truth values of the other logical connectives are obtained by the usual definitions.

Definition 5.9: *Let M be a structure, t be a term of sort s and θ be an assignment for M. Then $M(t\theta) \in s^M$ denotes the value of t in the structure M under the assignment θ. $M(t\theta)$ is inductively defined as follows:*

i) $M(x\theta) := \theta(x)$.

ii) $M(f(t_1,\ldots,t_n)\theta) := f^M(M(t_1\theta),\ldots,M(t_n\theta))$.

Definition 5.10: *Let M be a structure, θ be an assignment for M, $m \in s^M$ and $x \in V_s$. $\theta\frac{m}{x}$ is defined as follows:*

$$\theta\tfrac{m}{x}(y) := \begin{cases} m & y = x \\ \theta(y) & y \neq x \end{cases}$$

Definition 5.11: *Let M be a structure, A be a formula and θ be an assignment for M. Then $M \models A\theta$ denotes the* truth value *of A in the structure M under the assignment θ. $M \models A\theta$ is inductively defined as follows:*

i) $M \models \top\theta$

ii) $M \models r(t_1,\ldots,t_n)\theta :\Longleftrightarrow (M(t_1\theta),\ldots,M(t_n\theta)) \in r^M$

iii) $M \models \neg A\theta :\Longleftrightarrow M \not\models A\theta$

iv) $M \models (A \wedge B)\theta :\Longleftrightarrow M \models A\theta \wedge M \models B\theta$

v) $M \models (\exists x\ A)\theta :\Longleftrightarrow \exists m \in s^M(M \models A\frac{m}{x}\theta)$

Definition 5.12: *Let $A \equiv B$ denote that $M \models A\theta \Longleftrightarrow M \models B\theta$ for all structures M and all assignments θ for M. We define:*

i) $\bot \equiv \neg\top$

ii) $A \vee B \equiv \neg(\neg A \wedge \neg B)$

iii) $A \rightarrow B \equiv \neg A \vee B$

iv) $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

v) $\forall x\ A \equiv \neg\exists x\ \neg A$

The previous definitions seem rather tedious. But they are essential for the way predicate logic deals with arbitrary structures. There is always a clear distinction between the objects that we are speaking of and the language itself. The distinction is important to get results concerning the expressiveness of languages. The distinction is also an abstraction from the objects towards the language which is nothing other than a quantification over the structures. The quantification has some benefits for computer science if one identifies structure with implementation and language with specification.

Definition 5.13: *Let M be a structure and let A be a formula. Then $M \models A$ denotes that M is a* model *of A. $M \models A$ is defined as follows:*

$$M \models A :\Longleftrightarrow \forall\theta(M \models A\theta)$$

Definition 5.14: *Let M be a structure and let T be a set of formulas. Then $M \models T$ denotes that M is a* model *of T. $M \models T$ is defined as follows:*

$$M \models T :\Longleftrightarrow \forall A \in T(M \models A)$$

Definition 5.15: *Let K denote the class of all structures that have certain predefined domains, functions and built-ins. Let T be a set of formulas and A be a* *formula. Then* $T \vdash_K A$ *denotes that* T entails A. $T \vdash_K A$ *is defined as follows:*

$$T \vdash_K A :\Longleftrightarrow \forall M \in K(M \models T \rightarrow M \models A)$$

5.2 Rules and their Completion

In this section rules and their completion are discussed. The elements of a ProQuel knowledge base are not allowed to be arbitrary formulas but must have the form of a rule. In the following it is shown that rules are only able to describe negative knowledge. To obtain negative knowledge predicate completion is introduced which may introduces inconsistencies in some cases.

The positive and negative knowledge of a theory T are the formulas A with $T \vdash_K A$ and $T \vdash_K \neg A$ respectively. This conception belongs to the proof theoretic view which we will adopt in the first part of this section. In the last part of this section a shift to the model theoretic view will happen as an algebraic characterization of the models of rules and their completion by means of the program operator will be given. The characterisation is used in the development of the standard model for stratified knowledge bases in the next section.

A rule has the form $P \leftarrow A$ where P, called the *head*, is a predicate atom and A, called the *body*, is an arbitrary formula. The body of a rule is not restricted to a particular form of formulas. On the other hand, in the head of a rule, no built-ins, no logical connectives and no quantifiers are permitted.

Definition 5.16: *A formula of the form* $p(t_1, \ldots, t_n)$ *where* $p \in P$ *is called a* predicate atom. *A formula of the form* $P \leftarrow A$ *where* P *is a predicate atom and* A *is an arbitrary formula is called a* rule.

Example 5.17: *The following formulas are rules:*

$likes(jack, jill) \leftarrow \top$
$knows(X, Y) \leftarrow likes(X, Y)$
$flight(X, Y) \leftarrow \exists Z(flight(X, Z) \wedge flight(Z, Y))$

The following formulas are not rules:

$X = Y \leftarrow murder(X) \wedge murder(Y)$
$\forall Y \exists X\, likes(X, Y)$
$frozen(X) \vee broken(X) \leftarrow damaged(X)$

Rules of the form $P \leftarrow \top$ are called *facts*. In the following facts are abbreviated by P alone. The restriction to rules has a severe impact on the expressiveness of the knowledge base. Rules are able to describe positive atomic knowledge only. It means that we cannot deduce the negation of a predicate atom from a set of rules. As a consequence rules are always consistent but not complete in general.

Proposition 5.18: *Let* D *be a set of rules.*
Then $D \not\vdash_K \neg P$ *for every predicate atom* P.

Proof: Take the structure $M \in K$ with $p^M = s_1^M \times \ldots \times s_n^M$ for all $p \in P$ with

$\rho(p) = s_1 \times \ldots \times s_n$. Clearly $M \models D$ holds for any set of rules, but $M \not\models \neg P$ for all predicate atoms P. $\square$

Example 5.19: *Consider the following set of rules:*

$even(0)$
$even(X+2) \leftarrow even(X)$

We can deduce $even(n)$ *for even numbers* n *but we can not deduce* $\neg even(n)$ *for odd numbers* n*. There is not enough information about being not an even number.*

Example 5.20: *Consider the following set of rules:*

$smoker(john)$
$sport(john)$
$sport(jack)$
$sane(X) \leftarrow sport(X) \wedge \neg smoker(X)$

This example is rather pathological since we can neither deduce $sane(jack)$ *nor* $\neg sane(john)$*. Because there is no information about not being a smoker there is no information about being sane.*

If one wants to have negative knowledge one has to add formulas stating negative knowledge. For example if one knows that jack is a non-smoker then one must add the formula $\neg smoker(jack)$ in example 5.20. Explicit negative knowledge has some drawbacks. First of all, as proposition 5.18 states, to formalize negative knowledge one must go beyond rules. Second, in databases, it is not convenient to have explicit negative knowledge. For example in a flight database, flights that are not existent are simply not recorded. Accordingly, in deductive databases, one wants to stick to rules and nevertheless obtain negative knowledge.

The closed-world assumption (CWA) and circumscription are attempts to add knowledge by default to a theory. The CWA extends the theory by negated ground atoms that are determined by derivabillity from the original theory. Circumscription is a syntactic translation that results in a second-order theory [Lif85], [McC80], [Rei78]. Both attempts apply to arbitrary sets of formulas but introduce higher-order concepts. In the following we will rely on another attempt called predicate completion that is purely first-order. Predicate completion is only applicable to rules and consists of a simple syntactic translation.

The predicate completion of a finite set of rules D, which results in a set of formulas comp(D), was first introduced by [Cla78]. The completion essentially replaces the $\leftarrow$ symbol of the rules by the $\leftrightarrow$ symbol. If $P \leftarrow A_1, \ldots, P \leftarrow A_n$ are the rules for p then the completion will essentially contain one formula $P \leftrightarrow A_1 \vee \ldots \vee A_n$ for p. The elaborated definition is given in the following:

Definition 5.21: *Let* $p \in P$ *be a predicate with* $\rho(p) = s_1 \times \ldots \times s_n$*. The first step is to transform a rule:*

$p(t_1, \ldots, t_n) \leftarrow A$

into the rule:

$$p(x_1, \ldots, x_n) \leftarrow \exists y_1 \ldots \exists y_m(x_1 =_{s_1} t_1 \wedge \ldots \wedge x_n =_{s_n} t_n \wedge A)$$

where $x_1 \in V_{s_1}, \ldots, x_n \in V_{s_n}$ *are fresh variables and* $y_1, \ldots, y_m$ *are the free variables of* $t_1, \ldots, t_n$ *and* A*. Now suppose this transformation is made for each rule in the definition of* p *such that we obtain* $k \geq 0$ *transformed rules of the form:*

$$p(x_1, \ldots, x_n) \leftarrow A_1$$

$$\ldots$$

$$p(x_1, \ldots, x_n) \leftarrow A_k$$

The completed definition of p *is then the closed formula:*

$$\forall x_1 \ldots \forall x_n(p(x_1, \ldots, x_n) \leftrightarrow A_1 \vee \ldots \vee A_k)$$

The completion $comp(D)$ *of* D *is the collection of the completed definitions of the predicates* $p \in P$*.*

Note that the completed definition of a predicate p with no rules is the formula $\forall x_1 \ldots \forall x_n(p(x_1, \ldots, x_n) \leftrightarrow \bot)$. The completion clearly introduces new knowledge because one can now reason about negative knowledge. The completion also preserves the knowledge of the original set of rules. All the knowledge available in D is still available in $comp(D)$:

Proposition 5.22: *Let* D *be a set of rules. Let* A *be a formula. Then* $D \vdash_K A$ *implies* $comp(D) \vdash_K A$*.*

Proof: If $M \models comp(D)$ then $M \models P \leftrightarrow A_1 \vee \ldots \vee A_n$ for all completed definitions. Hence $M \models P \leftarrow A_i$ for $i = 1 \ldots n$ and therefore $M \models D$. □

Example 5.23: *Consider the completion of example 5.19:*

$$even(X) \leftrightarrow X = 0 \vee \exists Y(X = Y + 2 \wedge even(Y))$$

We can now deduce $\neg even(n)$ *for odd numbers* n*.*

Example 5.24: *Consider the completion of example 5.20:*

$$smoke(X) \leftrightarrow X = john$$
$$sport(X) \leftrightarrow X = john \vee X = jack$$
$$sane(X) \leftrightarrow sport(X) \wedge \neg smoke(X)$$

We can now deduce $sane(jack)$ *and also* $\neg sane(john)$*.*

The behaviour of the completion in the previous examples is very satisfying. Unfortunately this is not always the case. It is possible that the completion is incomplete and even inconsistent.

Example 5.25: *We want to reason about the transitive closure path of a relation arc. Consider the following rules:*

$arc(1,2)$
$arc(2,1)$
$path(X,Y) \leftarrow arc(X,Y)$
$path(X,Y) \leftarrow path(X,Z) \wedge arc(Z,Y)$
$nopath(X,Y) \leftarrow \neg path(X,Y)$

We cannot deduce $nopath(3,1)$.
Now consider the completion:

$arc(X,Y) \leftrightarrow (X=1 \wedge X=2) \vee (X=2 \wedge X=1)$
$path(X,Y) \leftrightarrow arc(X,Y) \vee \exists Z(path(X,Z) \wedge arc(Z,Y))$
$nopath(X,Y) \leftrightarrow \neg path(X,Y)$

We can still not deduce $nopath(3,1)$ *because* M *with* $path^M = \{(1,2),(2,1),(1,1),(2,2),(3,1),(3,2)\}$ *is a model. The completion is only able to state that path is a transitive relation that contains arc but it is not able to state that path is the least relation with that property.*

Example 5.26: *Consider the famous barber paradox. A barber of a small village must shave those who are not able to shave themselves:*

$shave(barber, X) \leftarrow \neg shave(X,X)$

If the barber does not shave himself he acts against the law in disregarding the material implication. On the other hand if the barber shaves himself he does not act against the law, because the law makes no statement whatever about people the barber may not shave. Consider the completion:

$shave(barber, X) \leftrightarrow \neg shave(X,X)$

Now it is impossible for the barber to conform with the law. The completion is inconsistent and we can deduce $shave(barber, barber)$ *and* $\neg shave(barber, barber)$.

To end this section we now turn to the algebraic characterisation of the models of rules and their completion. The characterisation is based on the program operator T_D of a set of rules D. The program operator not only affords another insight to the interplay of rules and their completion, but is also useful in defining the standard model and in the implementation of deductive database. The following definition and propositions are essentially found in [Llo86, p.81] in the context of normal programs.

Definition 5.27: *Let* D *be a set of rules. The program operator* T_D *is defined to map structures* M *to structures* $T_D(M)$ *with the same predefined domains, functions and built-ins such that for all* $p \in P$:

$$p^{T_D(M)} := \{(M(t_1\theta), \ldots, M(t_n\theta)) \mid p(t_1, \ldots, t_n) \leftarrow A \in D \wedge M \models A\theta\}$$

Proposition 5.28: *Let* D *be a set of rules and* M *be a structure. Then the following holds:*

$$M \models D \Longleftrightarrow T_D(M) \subseteq M$$

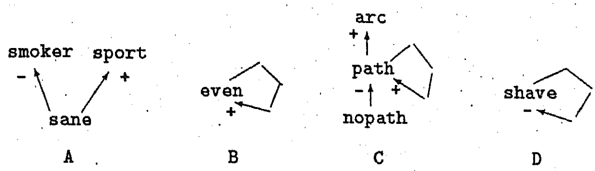

Figure 5.1: Dependency Graph

Proposition 5.29: *Let D be a set of rules and M be a structure. Then the following holds:*

$$M \models comp(D) \Longleftrightarrow T_D(M) = M$$

5.3 The Standard Model

In the previous section it has been shown that rules are not able to represent negative information. For this purpose the completion has been introduced which gives more completeness but may also cause inconsistencies. In the following we will take a closer look at the models of the completion for some syntactical classes of rules. In particular we will discuss the classes of hierarchical, definite and stratified rules that all do have consistent completions. There are still some other interesting classes that are not discussed here. For a brief survey see [Sat90]. In the end of this section we will concentrate on the standard model of stratified rules.

The classes hierarchical, definite and stratified are all defined by means of the dependency graph. The dependency graph consists simply of positive and negative arcs between predicates. Figure 5.1 shows the dependency graphs of the examples 5.20, 5.19, 5.25 and 5.26 from the previous section.

Definition 5.30: *Let A be a formula and $p \in P$ a predicate. The predicate p* occurs positively (negatively) *in A if there is a sub-formula $p(t_1, \dots, t_n)$ of A that is in the scope of an even (odd) number of negations.*

Definition 5.31: *Let D be a set of rules and $p, q \in P$ predicates. The predicate p* depends positively (negatively) *on q if there is a rule $p(t_1, \dots, t_n) \leftarrow A$ in D and q occurs positively (negatively) in A.*

Hierarchical rules are now defined as rules that do not contain recursion whereas definite rules are defined as rules that may contain recursion but no negation. Graph A is hierarchical but not definite. On the other hand graph B is definite but not hierarchical. Finally examples C and D are neither hierarchical nor definite. Stratified rules are a genuine combination of definite and hierarchical rules. Stratified rules may contain recursion and negation although with the restriction that recursion may not go through negation. A, B and C are all stratified but D is not stratified.

Definition 5.32: *Let D be a set of rules. D is called* hierarchical *if and only if there is a level mapping $n : P \rightarrow I\!N$ such that $n(p) > n(q)$ whenever p depends on q. In hierarchical rules recursion is not allowed.*

Definition 5.33: *Let D be a set of rules. D is called* definite *if and only if there are no p, q such that p depends negatively on q. In definite rules negation is not allowed.*

Definition 5.34: *Let D be a set of rules. D is called* stratified *if and only if there is a level mapping $n : P \rightarrow I\!N$ such that $n(p) \geq n(q)$ whenever p depends positively on q and $n(p) > n(q)$ whenever p depends negatively on q. In stratified rules recursion through negation is not allowed.*

Lets take a closer look to the fix-point models of these classes. By fix-point models we mean structures with predefined domains, functions and built-ins that are models of the completion, i.e. $M \in K$ and $M \models comp(D)$. The first observation will be that hierarchical rules have exactly one fix-point model. In hierarchical rules the user-definable predicates depend only on relations previously defined. By induction they depend only on the built-ins and are therefore uniquely determined.

Lemma 5.35: *Let M_1 and M_2 be two structures with the same domains, functions and built-ins. If $q^{M_1} = q^{M_2}$ whenever p depends on q then $p^{T_D(M_1)} = p^{T_D(M_2)}$.*

Proposition 5.36: *Let D be a hierarchical rule set. Then there exists exactly one structure $M \in K$ with $M \models comp(D)$.*

Proof: Using a level mapping and lemma 5.35 one can inductively construct with the help of T_D an $M \in K$ with $M \models comp(D)$. Now assume $M_1 \models comp(D)$ and $M_1 \models comp(D)$ for $M_1, M_2 \in K$. Using again a level mapping and lemma 5.35 one can show that $M_1 = M_2$ by induction. □

Very important too is the class of definite rules. Definite rules have the pleasant property that their program operator is monotonic. According to [Tas55] the fix-point models form a lattice. Hence definite rules do have a whole range of fix-point models that are all set up in a lattice.

Lemma 5.37: *Let M_1 and M_2 be two structures with the same domains, functions and built-ins. If $q^{M_1} \subseteq q^{M_2}$ whenever p depends positively on q and $q^{M_1} \supseteq q^{M_2}$ whenever p depends negatively on q, then $p^{T_D(M_1)} \subseteq p^{T_D(M_2)}$.*

Proposition 5.38: *Let D be a definite rule set. Then the structures $M \in K$ with $M \models comp(D)$ form a lattice.*

Proof: According to lemma 5.37 the program operator T_D is monotonic. Apply the fix-point theorem of [Tas55]. □

Stratification was first introduced by [ABW86] in the context of quantifier free rules. [ABW86] showed that stratified quantifier free rules have a Herbrand fix-point model. The result is easily extended to our context where quantifiers and arbitrary predefined domains, functions and built-ins are allowed. The fix-point models of stratified rules do not form a lattice anymore. All that can be said is that stratified rules have at least one fix-point model.

Proposition 5.39: *Let D be a stratified rule set. Then there is a structure $M \in K$ with $M \models comp(D)$.*

Proof: According to lemma 5.35 and 5.37 the program operator T_D is monotonic in each level for a level mapping. By virtue of the fix-point theorem in [Tas55] we can construct a fix-point at each level. Iterating this process the iterated fix-point IF_D is obtained which is clearly a fix-point model. □

The *iterated fix-point* was first obtained in [ABW86] in the context of quantifier free rules. [ABW86] showed also that the iterated fix-point is independent of the used level mapping. The notion of iterated fix-point should not be confused with the notion of fix-point iteration. The latter describes the way a single fix-point could be generated whereas the former describes the multiple generation of fix-points. Fix-point iteration is only applicable to continuous program operators. Because arbitrary formulas are allowed in our context the program operator is not necessarily continuous and we have to rely on the fix-point theorem of [Tas55] which is a pure existence theorem.

In choosing a fix-point model for one of these classes we are faced with the problem that not all classes uniquely determine a fix-point model. Except for hierarchical rules there can be many fix-point models. In adding the constraint that the fix-point model should be minimal we can single out the least fix-point for definite rules. Unfortunately stratified rules can have many different minimal fix-point models and therefore no least fix-point model at all.

Example 5.40: *Consider the definite rules $D = \{p \leftarrow p\}$. The completion $comp(D) = \{p \leftrightarrow p\}$ has the two models $M_1(p) = 0$ and $M_2(p) = 1$. M_1 is the least model.*

Example 5.41: *Consider the stratified rules $D = \{p \leftarrow p, q \leftarrow \neg p\}$. The completion $comp(D) = \{p \leftrightarrow p, q \leftrightarrow \neg p\}$ has the two models $M_1(p) = 0, M_1(q) = 1$ and $M_2(p) = 1, M_2(q) = 0$. M_1 and M_2 are both minimal.*

We have to impose an additional constraint for stratified rules. The constraint will be *preservation* which means that one can add new predicates without disturbing the meaning of old ones. Preservation is highly desirable in the development of knowledge bases because one wants to use previously written and tested predicates without influencing them.

Example 5.42: *If we want to have a minimal and preserving fix-point model then we cannot assign the fix-point model $M_2(p) = 1, M_2(q) = 0$ to the stratified rules $D = \{p \leftarrow p, q \leftarrow \neg p\}$. D is an extension of $D' = \{p \leftarrow p\}$ which has the least model $M(p) = 0$. M_2 is not preserving because $M_2(p) \neq M(p)$.*

Under the two constraints minimality and preservation there is exactly one possible fix-point model for stratified rules. The unique fix-point model is the same as the iterated fix-point IF_D. The iterated fix-point is therefore called the *standard*

Class	Unrestricted	Minimal	Minimal and Preserving
Hierarchical	Unique	Unique	Unique
Definite	Not unique	Unique	Unique
Stratified	Not unique	Not unique	Unique

Table 5.1: Uniqueness of Fix-Point Models

model for stratified rules. Table 5.1 shows the influence of successively imposing the constraints fix-point model, minimality and preservation to the basic classes. The standard model is the only minimal and preserving fix-point model semantic for stratified rules.

Definition 5.43: *Let ϕ be a mapping from rules D to structures $\phi(D) \in K$. ϕ is called a fix-point model semantic iff $\phi(D) \models comp(D)$.*

Definition 5.44: *Let ϕ be a fix-point model semantic. ϕ is called minimal when there is no M with $M \models comp(D)$ and $M \subseteq \phi(D)$ for every set of rules D.*

Definition 5.45: *Let ϕ be a fix-point model semantic. ϕ is called preserving when $\phi(D) = \phi(D')|P$ for every set of rules D and D' such that D' is an extension of D.*

Proposition 5.46: *The standard model is the one and only minimal and preserving fix-point model semantic for stratified rules.*

Proof: Clearly IF_D is a minimal and preserving fix-point model semantic. Now assume that ϕ_1 and ϕ_2 are two minimal and preserving fix-point model semantics. By induction on a level mapping it is easy to show that $\phi_1(D) = \phi_2(D)$ for every set of rules D because of preservation and minimality. □

5.4 Computational Tractability

The true sentences of the standard model are neither decidable nor semi-decidable. Therefore, the class of accepted queries and rules has to be further restricted to assure their computational tractability. In the following the expressiveness of the iterated fix-point IF_D is first discussed. Then the notion of allowedness is introduced. It is a syntactical criterion that assures the computational tractability of queries if a standard model with finite predicates exists.

In the following it is demonstrated that every computational function can be defined by an appropriate set of definite rules. The demonstration follows [Llo86, p.53] where essentially the same result is shown in a proof theoretic setting. In particular we will show that every partial recursive function f can be defined in the standard model semantic by a set $D(f)$ of definite rules. The standard model must have the domain of natural numbers $\mathbb{N}$, the zero constant $0 : \mathbb{N}$ and the successor function $+1 : \mathbb{N} \rightarrow \mathbb{N}$.

Definition 5.47: *Let f be a partial recursive function. The set $D(f)$ of definite rules is inductively defined as follows:*

i) Zero function $f(x) = 0$. *Define* $D(f)$ *to be the following rule:*

$p_f(x, 0)$

ii) Successor function $f(x) = x + 1$. *Define* $D(f)$ *to be the following rule:*

$p_f(x, x + 1)$

iii) Projection functions $f(x_1, ..., x_n) = x_j$. *Define* $D(f)$ *to be the following rule:*

$p_f(x_1, ..., x_n, x_j)$

iv) Composition $f(x_1, ..., x_n) = h(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$. *Define* $D(f)$ *to be* $D(h) \cup D(g_1) \cup ... \cup D(g_m)$ *plus the following rule:*

$p_f(x_1, ..., x_n, x) \leftarrow p_{g_1}(x_1, ..., x_n, y_1) \wedge ... \wedge p_{g_m}(x_1, ..., x_n, y_m) \wedge p_h(y_1, ..., y_m, x)$

v) μ-Recursion $f(x_1, ..., x_n) = \mu y(g(x_1, ..., x_n, y) = 0)$. *Define* $D(f)$ *to be* $D(g)$ *plus the following three rules:*

$p_f(x_1, ..., x_n, x) \leftarrow p_g(x_1, ..., x_n, 0, z) \wedge h_f(x_1, ..., x_n, 0, z, x)$

$h_f(x_1, ..., x_n, y, 0, y)$

$h_f(x_1, ..., x_n, y, z + 1, x) \leftarrow p_g(x_1, ..., x_n, y + 1, t) \wedge h_f(x_1, ..., x_n, y + 1, t, x)$

Lemma 5.48: *Let* f *be a partial recursive function. Let* $IF_{D(f)}$ *be the standard model of* $D(f)$ *over the natural numbers. Then for all* $k_1, \ldots, k_n, k \in I\!N$ *the following holds:*

$$f(k_1, \ldots, k_n) \doteq k \iff IF_{D(f)} \models p_f(k_1, \ldots, k_n, k)$$

Proposition 5.49: *The true sentences in* IF_D *are neither decidable nor semi-decidable.*

Proof: By a reduction to the halting problem the true sentences in IF_D are not recursive. Consider the sentences $\forall x \exists y\ p_f(x, y)$ which denotes the totality of the partial recursive function f. As it is well known the code of the total recursive functions is not recursively enumerable. Hence the true sentences in IF_D are even not recursively enumerable in general. □

The notion of allowedness is often introduced in a proof theoretic context to assure domain independence [TS88]. In our case allowedness is used to tackle the problem of *safe* queries, i.e. the finiteness of the answer set. Here we will present an allowedness notion that is slightly different from the one in [TS88]. The notion is based on a procedural reading of formulas from left to right. Every formula A is regarded a many-valued function $M[A\theta]$ that produces a set of assignments for a given assignment θ and a given structure M. The intention is that $M[A\theta]$ denotes all the extensions ϕ of θ such that $M \models A\phi$.

Definition 5.50: *Let* $\phi : Y \rightarrow U$. *Then* $\phi_X : X \rightarrow U$ *denotes the* restriction *of* ϕ *to* $X \subseteq Y$, *i.e.* $\phi_X(x) = \phi(x)$ *for* $x \in X$.

Definition 5.51: *Let* A *be a formula,* M *be a structure and* $\theta : X \rightarrow U$. *Then the* associated function $M[A\theta]$ *is defined as follows:*

$$M[A\theta] := \{\phi : Y \to U \mid \phi_X = \theta \wedge M \models A\phi\}, \quad Y = X \cup var(A)$$

Example 5.52: *Consider* $A = p(X,Y)$, M *with* $p^M = \{(1,2),(1,3),(2,3)\}$ *and* $\theta = \{X/1\}$. *Then* $M[A\theta] = \{\phi_1, \phi_2\}$ *with* $\phi_1 = \{X/1, Y/2\}$ *and* $\phi_2 = \{X/1, Y/3\}$.

Computing the answer set of a query Q to a database D under the standard model semantic clearly amounts to the problem of determining $IF_D[Q\emptyset]$ where $\emptyset$ is the empty assignment. Likewise safety means that $M[Q\emptyset]$ is finite for a structure M with finite predicates (p^M finite for all $p \in P$). Hence understanding the safety of queries means understanding the associated function. The associated function has the following properties:

Definition 5.53: *Let* $S \subseteq \{\theta : X \to U\}$. *Then* $S^Y := \{\phi : Y \to U \mid \phi_X \in S\}$ *denotes the* extension *of* S *to* $Y \supseteq X$.

Lemma 5.54: *Let* M *be a structure and* $\theta : X \to U$. *The following holds:*

i) $M[A \wedge B\theta] = \bigcup_{\phi \in M[A\theta]} M[B\phi]$

ii) $M[A \vee B\theta] = M[A\theta]^Y \cup M[B\theta]^Y, \quad Y = X \cup var(A) \cup var(B)$

iii) $M[\neg A\theta] = \theta^Y \setminus M[A\theta], \quad Y = X \cup var(A)$

iv) $M[\exists x\, A\theta] = \{\phi_Y \mid \phi \in M[A\theta]\}, \quad Y = X \cup var(A) \setminus \{x\} \quad (x \notin X)$

Let us examine the safety of the associated function by means of lemma 5.54 for an infinite domain U. Note that for infinite domains U the set S^Y is infinite in case that $Y \neq X$ and $S \neq \emptyset$. Now clearly the conjunction $M[(A \wedge B)\theta]$ is safe in case that $M[A\theta]$ is safe and that all $M[B\phi]$ are safe for $\phi \in M[A\theta]$. The disjunction $M[(A \vee B)\theta]$ is safe in case that $M[A\theta]$ and $M[B\theta]$ are safe and that $X \cup var(A) = X \cup var(B)$. The existential quantification $M[(\exists x\, A)\theta]$ is safe in case that $M[A\theta]$ is safe. Finally the negation is safe in case that $M[A\theta]$ is safe and that $var(A) \subseteq X$. We are now ready to state the allowedness criterion:

Definition 5.55: *Let* t *be a term and* X *be a set of variables.* $in(t,X)$ *and* $out(t,X)$ *are defined as follows:*

i) $in(t,X) :\Longleftrightarrow var(t) \subseteq X$

ii) $out(t,X) :\Longleftrightarrow t \in V \vee in(t,X)$

Definition 5.56: *Let* A *be a formula and* X *be a set of variables. Then* $ok(A,X)$ *denotes that* A *is allowed for input* X. $ok(A,X)$ *is defined as follows:*

i) $ok(A \wedge B, X) :\Longleftrightarrow ok(A,X) \wedge ok(B, X \cup var(A))$

ii) $ok(A \vee B, X) :\Longleftrightarrow ok(A,X) \wedge ok(B,X) \wedge X \cup var(A) = X \cup var(B)$

iii) $ok(\neg A, X) :\Longleftrightarrow ok(A,X) \wedge var(A) \subseteq X$

iv) $ok(\exists x\, A, X) :\Longleftrightarrow ok(A, X \setminus \{x\})$

v) $ok(s = t, X) :\Longleftrightarrow (in(s,X) \wedge out(t,X)) \vee (in(t,X) \wedge out(s,X))$

vi) $ok(p(t_1, \ldots, t_n), X) :\Longleftrightarrow out(t_1, X) \wedge \ldots \wedge out(t_n, X)$

Proposition 5.57: *Let M be a structure without extra built-ins, $\theta : X \to U$ a partial assignment for M and A be a formula such that $ok(A, X)$ holds. Then the following holds:*

$$M \textit{ has finite predicates} \Longrightarrow M[A\theta] \textit{ is finite}$$

Proof: Use induction on A. The induction step is easily verified by lemma 5.54. For the base case use the fact that $M(t\theta)$ is defined for $var(t) \subseteq X$. □

Allowedness for input can now be used to assure the safety of queries. Assume that the predicates of IF_D are all finite, then from $ok(Q, \emptyset)$ it follows that $IF_D[Q\emptyset]$ is finite too. It is also easy to see that additionally the answer set $IF_D[Q\emptyset]$ is totally recursive in case that the predefined functions are all totally recursive. The problem that rests is the problem of computing IF_D. We don't know yet how to compute IF_D and whether the predicates will be finite. Therefore the notion of allowed rules is introduced.

Definition 5.58: *Let $P \leftarrow A$ be a rule. $P \leftarrow A$ is called* allowed *if and only if $var(P) \subseteq var(A)$ and $ok(A, \emptyset)$.*

Lemma 5.59: *Let D be a finite set of allowed rules and M be a structure without extra built-ins. Then the following holds:*

$$M \textit{ has finite predicates} \Longrightarrow T_D(M) \textit{ has finite predicates}$$

The application of the program operator T_D of allowed rules to a structure with finite predicates gives a structure with finite predicates. Now it is also easy to see that T_D is totally recursive for structures with finite predicates in case that the predefined functions are all totally recursive. Hence it is possible to effectively iterate T_D and therefore if a finite fix-point exists it will be reached in a finite number of iteration steps. The allowedness notion presented so far is essentially used in ProQuel. It allows ProQuel to compute the standard model if a finite one exists. Through the notion of allowedness no computational power is lost:

Proposition 5.60: *Allowed rules with at least the zero function and the successor function are computational complete.*

Proof: We show that for every partial resursive function f there is a definite set of allowed rules $\tilde{D}(f)$. The set $\tilde{D}(f)$ is obtained from $D(f)$ in definition 5.47 by changing every rule $P \leftarrow A$ to:

$P \leftarrow nat(x_1) \wedge \ldots \wedge nat(x_n) \wedge A$

Where $\{x_1, \ldots, x_n\} = var(P) \setminus var(A)$ and by adding the following two rules:

$nat(0)$

$nat(x+1) \leftarrow nat(x)$

Clearly the rules are all allowed and $IF_{D(f)} = IF_{\tilde{D}(f)}|P$. □

Chapter 6

Bibliography

[ABW86] *Apt, K.R., Blair, H. and Walker, A.*: Towards a Theory of Declarative Knowledge. IBM Res. Report RC 11681, April 1986

[ASU86] *Aho, A. V., Sethi, R. and Ullman, J. D.*: Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company, 1986

[Ber91] *Bertschinger, M.H.*: Eine temporale deduktive Datenbank. Diploma Thesis, ETH Zürich, Abt IIIC, 1991

[Bur89] *Burse, J.*: Inkrementeller Konsistenztest in deduktiven Datenbanken. Diploma thesis, ETH Zürich, Abt IIIC, 1989

[CGT90] *Ceri, S., Gottlob, G. and Tanca, L.*: Logic Programming and Data- bases. Springer Verlag, 1990

[CM84] *Clocksin, W. F. and Mellish, C. S.*: Programming in Prolog, second Edition. Springer Verlag, 1984

[Cla78] *Clark, K.L.*: Negation as Failure. Logic and Data Bases, Gallaire, H. and Minker, J. (eds), Plenum Press, New York, p. 293-322, 1978

[Hel91] *Helbing, R.*: Implementierung eines deduktiven Datenbanksystems in Prolog. Diploma Thesis, ETH Zürich, Abt IIIC, 1991

[Kel91] *Keller, J.*: Handhabung mengenwertiger Attribute durch ein deduktives Datenbanksystem. Diploma Thesis, ETH Zürich, Abt IIIC, 1991

[Lif85] *Lifschitz, V.*: Closed-World Databases and Circumscription. Artificial Intelligence, v. 27, p. 229-235, 1985

[Llo85] *Lloyd, J.W.*: A Basis for Deductive Database Systems. The Journal of Logic Programming, v. 2, p. 93-109, 1985

[Llo87] *Lloyd, J.W.*: Foundations of Logic Programming. 2nd Extended Editon, Springer Verlag, 1987

[Mel84] *Melhorn, K.*: Graph Algorithms and NP-Completeness. Monographs on Theoretical Computer Science, Springer Verlag, 1984

[MWW89] *Marti, R., Wieland, C., and Wüthrich, B.*: Adding Inference to a Relational Database Management System. Proc. BTW, 1989

[McC80] *McCarthy, J.*: Circumscription - A Form of Non-Monotonic Reasoning. Artificial Intelligence, v. 13, p. 27-39, 1980

[Rei78] *Reiter, R.*: On Closed World Data Bases. Logic and Databases, H. Gallaire and J. Minker (eds), Plenum Press, New York, 1978

[Sat90] *Sato, T.*: Complete Logic Programs and their Consistency. The Journal of Logic Programming, v. 9, p. 33-44, 1990

[SS86] *Sterling, L. and Shapiro, E.*: The Art of Prolog. The MIT Press, 1986

[Taj72] *Tarjan, R.*: Depth-First Search and Linear Graph Algorithm. SIAM Journal on Computing, p. 157, 1972

[Tas55] *Tarski, A.*: A Lattice-theoretical Fix-point Theorem and its Applications. Pacific J. Math. v. 5, p. 285-309, 1955

[TS88] *Topor, R.W., and Sonenberg, E.A.*: On Domain Independent Data- bases. Foundations of Deductive Databases and Logic Programming, Minker, J. (eds), Morgan Kaufmann Publishers Inc., 1988

[Wan52] *Wang, H.*: Logic of Many Sorted Theories. The Journal of Symbolic Logic, v. 17, 1952

[Wal84] *Walther, C.*: Schubert's Steamroller - A Case Study in Many-Sorted Resolution. Interner Bericht 5/84, Universität Karlsruhe, 1984

[Wüt91] *Wüthrich,B.*: Large Deductive Databases with Constraints. Dissertation, ETH Zürich Nr. 9401, 1991

Gelbe Berichte des Departements Informatik

159	K. Gates	Using Inverse Iteration to Improve the Divide and Conquer Algorithm.
160	H. Mössenböck	Differences between Oberon and Oberon-2 The programming Language Oberon-2 (vergriffen)
161	S. Lalis	XNet: Supporting Distributed Programming in the Oberon Environment (vergriffen)
162	G. Weikum, C. Hasse	Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism
163	J. Nievergelt et al.	eXperimental geometrY Zurich: Software for Geometric Computation
164	G.H. Gonnet, H. Straub	The Dynamic Programming Algorithm as a Finite Automation
165	L. Adams, P. Arbenz	Towards a Divide and Conquer Algorithm for the Real Nonsymmetric Eigenvalue Problem
166	E. Margulis	N-Poisson Document Modelling Revisited
167	H.E. Meier	Schriftgestaltung mit Hilfe des Computers Typographische Grundregeln mit Gestaltungsbeispielen (neue erweiterte Auflage)
168	B. Heeb, I. Noack	Hardware Description of the Workstation Ceres-3
169	M. Bronstein	Formulas for Series Computations
170	P. Arbenz	Divide and Conquer Algorithms for the Bandsymmetric Eigenvalue Problem
171	K. Simon, P. Trunz	On Transitive Orientation
172	A. Rosenthal, Ch. Rich, M.H. Scholl	Reducing Duplicate Work in Relational Join(s): A Unified Approach
173	P. Schäuble, B. Wüthrich	On the Expressive Power of Query Languages
174	M. Brandis, R. Crelier M. Franz, J. Templ	The Oberon System Family
175	P. Scheuermann, G. Weikum, P. Zabback	Automatic Tuning of Data Placement and Load Balancing in Disk Arrays
176	H. Hinterberger J.C. French (eds.)	Proceedings of the Sixth International Working Conference on Scientific and Statistical Database Management