A Powerful Prolog to SQL Compiler

Christoph Draxler
CIS Centre for Information and Language Processing
Ludwig-Maximilians-Universität München
Wagmüllerstr. 23
D - 80538 München
Tel: +49 +89 211 06 64 (-60 secretariat, -74 fax)
draxler@cis.uni-muenchen.de

August 16, 1993

Abstract

This report describes the implementation of a compiler for translating Prolog database goals to SQL database queries. The implementation adheres to the guidelines of D.H.D. Warren [27] for writing compilers in Prolog.

The compiler is invoked by calling the goal

```
?- translate(ProjectionTerm, DatabaseGoal, SQLQuery).
```

All arguments are standard Prolog terms. `ProjectionTerm` specifies the attributes which are to be retrieved from the database, `DatabaseGoal` defines the selection restrictions and join conditions. `SQLQuery` is a term representing the SQL query. This term is transmitted to an SQL DBMS in a suitable form, e.g. as ASCII text.

The compiler extends previous work (e.g. [15, 19, 10, 9] in that it allows negation, the expression of arithmetic functions, and higher-order constructs such as grouping, sorting, and aggregate functions. Furthermore, the compiler is written in standard Edinburgh Prolog, and great care has been taken to write clean code. This not only makes the code readable, but also efficient and portable.

Benchmarks for different Prolog implementations are given. The appendix contains a summary of the improvements introduced by v. 1.1 of the compiler.

1 Introduction

A coupled system consists of a logic programming language connected to a relational database management system. In physically loosely coupled systems, both the logic programming language environment and the database management system run as separate processes communicating with each other through some communication channel [3]

In physically loosely coupled systems there exist three languages: the *logic programming language*, the *data manipulation language* of the database system, and the *database access language* through which the database is accessed from the logic programming language. A translation procedure may be necessary to translate database access requests into database queries.

Current physically loosely coupled systems consist of a Prolog environment connected to an external SQL DBMS. Thus, Prolog is the logic programming language and SQL is the data manipulation language. In general, the database access language a restricted sublanguage of Prolog.

In early systems, e.g. PROSQL [7], the database access language consisted of a built-in predicate the arguments of which contained an SQL query represented through a Prolog string. The query string was directly transmitted to the database system. The advantage of this approach is that it does not require any translation, and the full extent of SQL can be used. The big disadvantages are that the current state of the Prolog evaluation cannot be exploited to restrict queries, and application programmers are confronted with two distinct languages.

In more recent systems, the database access language consists of Prolog predicates which are translated to appropriate SQL queries. Educe [3], Bermuda [14], CGW [6] and many commercial Prolog systems (e.g. [1, 17]) feature single relation access, i.e. only one relation table can be accessed at any one time. Translating such access requests is straightforward, but queries can only be restricted through their argument values. In view access, as in [13, 9], or LogiQuel [18], multiple relation tables can be accessed simultaneously, allowing for join selectivity to restrict queries.

In all these systems, the database access language is defined through clauses in the program code, and this definition is static. Furthermore, view access is not powerful enough to capture the full range of SQL. Higher-order constructs such as grouping, aggregate functions, and sorting cannot be expressed.

In database set predicates [11], the database access language is defined through a projection term and a goal term. Both terms can be constructed at runtime, so that a fully dynamic query formulation, leading to maximally restrictive queries, is possible. In addition, because database set predicates are higher-order extensions of Prolog, the higher-order constructs of SQL can be exploited naturally.

The translation from Prolog to SQL has been sketched in the literature [15, 9], and a more detailed description, including the translation of negation, can be found in [19]. A compiler for translating ProQuel to different target languages, including SQL, POSTQUEL, and Prolog is presented in [5].

The compiler presented here extends previous work with the translation of arithmetic expres-

sions, aggregate functions including grouping, and sorting. Great care has been taken to write clean and readable code in standard Prolog. As a result, the compiler is efficient and portable.

The report is organized as follows: A brief definition of the database access language is given in section 2. Section 3 gives an overview over the compiler, which is then presented in more detail in section 4. Section 5 contains benchmarks, and section 6 concludes the report with a discussion of the compiler and an outlook towards future work.

2 Database access language

The database access language is a sublanguage of Edinburgh Prolog through which external databases are accessed.

2.1 Prolog

The basic Prolog datastructure is a term. A *term* is either a constant, a variable, or a term $t(a_1, ..., a_n)$ where t is the functor and the a_i (called *arguments*) are terms. A literal is a term of the form $t(a_1, ..., a_n)$. A literal preceded by the negation operator `not/1` is called a *negative* literal.

A *Prolog program* consists of an ordered set of program clauses. A *program clause* is either a

- *unit clause*: $H.$
- *rule*: $H :- B_1 \circ ... \circ B_n.$
- *goal*: $:- B_1 \circ ... \circ B_n.$

with H (the *head* of a clause) a positive literal, B_i (the *body*) positive or negative literals, and $\circ$ either the junctor "," (logical *and*) or ";" (logical *or*).

A *fact* is a ground unit clause, i.e. a unit clause the arguments of which do not contain variables. A *base goal* consists of one positive literal only. The standard *comparison operators* for terms are $=$, $@<$, $@>$ with their usual meaning. The predicate `is/2` takes as its input argument an arithmetic expression, and unifies the other argument with the result of the evaluation of the arithmetic expression. An *aggregate function term* is a literal of the form $agg(A, G, R)$ where A is a variable, G a goal with A one of its arguments, and R the result of the computation of the aggregate function agg for the variable A over the set of solutions of G.

The semantics of Prolog programs is defined through SLD resolution with negation as finite failure [8, 16]. The inference engine of Prolog evaluates goals from left to right. During the evaluation of a goal variables may become *instantiated*, i.e. they are *bound* to a value. The successful evaluation of a positive goal with variable arguments returns the bindings of these variable arguments.

For a *safe* evaluation, all arguments to comparison operators and negated goals, and all input arguments to the predicate `is/2`, must be bound [23, 16].

2.2 Database access language definition

A *database goal* is a base goal which is proved from data stored in external databases. A *database aggregate function term* is an aggregate function term the goal argument of which is a database goal.

A database access is defined through a projection term and a database goal term where

- *projection term* is a Prolog term
- *database goal term* is a sequence of literals $L_1, ..., L_n$, $n \geq 1$, connected through the junctors "," or ";" such that L_i is a
  - database goal,
  - database aggregate function term,
  - comparison operation, or the
  - arithmetic predicate `is/2`.

Variables in the database goal term are existentially quantified *implicitly* through the projection term, or *explicitly* through the quantification operator `^/2`. The bindings of existentially quantified variables are undone after the evaluation of a database access. Variables which are not existentially quantified are called *free* variables. Their bindings are returned as part of the result of a database evaluation.

2.3 Sample query

The database contains flight connections and airplane types. All flight connections are stored in the relation table *FLIGHT* with the attributes *FLIGHT_NO*, *DEPARTURE*, *DESTINATION*, *PLANE*, and all planes are stored in the relation table *PLANE* with the attributes *TYPE* and *SEATS*.

The following information is to be retrieved:

> "Retrieve from the database the destinations, plane types, and seats of flights leaving Munich with large planes, i.e. planes which have more than the average number of seats."

The projection term and database goal of the corresponding database access are

```
big_planes(munich,Dest,Type,Seats)
```

and

```
FNo^(
 flight(FNo,munich,Dest,Type),
 plane(Type,Seats),
 Seats > avg(S, T^plane(T,S))).
```

The database goal consists of a conjunction of three base goals: two positive calls to the database facts `flight/4` and `plane/2`, and the arithmetic comparison operation >, the right argument of which is an aggregate function term.

`munich` is a constant value. `Dest`, `Type`, and `Seats` are implicitly existentially quantified variables because they occur in both projection term and database goal. `FNo` is existentially quantified explicitly through the `^/2` operator. `S` and `T` are local to the aggregate function term, with `S` being the aggregate variable over which the aggregate function `avg` is to be computed on the database fact `plane/2`.

3 Overview over the Prolog to SQL compiler

The Prolog to SQL compiler consists of a Prolog meta-programming part for the tokenizer and the syntax analyzer, and a translator for the code generation and output.

The *lexical analysis* takes as input the Prolog terms `ProjectionTerm` and `DatabaseGoal`. Each variable in both terms is unified with a unique identifier of the form `var(VarId)` with `VarId` a positive integer. Constant values are placed in a term of the form `const(ConstantValue)`. These wrappers distinguish variables from constants and contribute to efficiency because argument indexing can be exploited (In the real implementation `'$var$'/1` and `'$const$'/1` are used to reduce the risk of name clashes with user defined goal names).

The *syntax analysis* transforms the database goal argument into a disjunctive normalized form in which the disjunction operators are moved to the front of the goal through applying the distributivity laws of the conjunction and the disjunction operator.

The *code generation* maps Prolog to SQL. Basically, a Prolog database predicate corresponds to an SQL relation table or view, with the predicate functor mapped to the table or view name, and each argument position mapped to an attribute name. Database goals are translated according to the following rules [19, 9, 11]:

- Disjunctive goals translate to distinct SQL queries connected through the UNION operator.
- Goal conjunctions translate to joins.
- Negated goals translate to negated EXISTS subqueries.
- Existentially quantified variables with single occurrence are not translated.
- Free variables translate to grouping attributes.
- Shared variables in goals translate to equi-join conditions.
- Constants translate to equality comparisons of an attribute and the constant value.

Comparison operations, arithmetic expression terms, and aggregate function terms translate to SQL comparison operations, arithmetic functions over attributes, and aggregate function

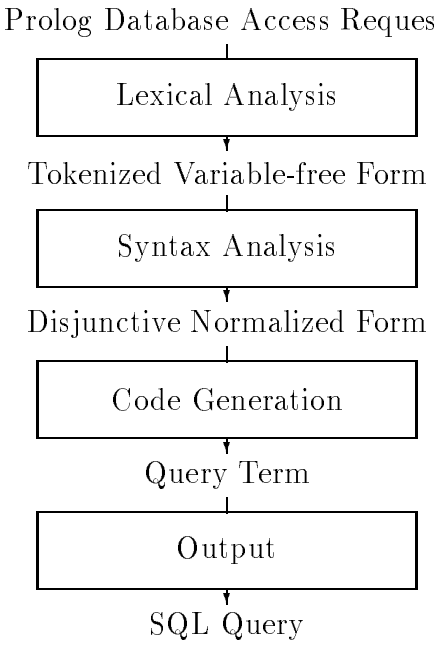


Fig 1: Compilation phases

(sub-)queries respectively. The result of the code generation is a query term which is a Prolog term representation of the SQL query corresponding to the projection term and database goal.

The *output* phase prints the query term in the correct SQL syntax format to an output device, e.g. a terminal, a file, or a communication channel.

Fig. 1: (adapted from D.H.D. Warren's paper on writing compilers in Prolog [27]) displays the compilation phases and the output of each phase.

4 The compiler implementation

The top level predicate `translate/3` reflects the compilation phases, as shown in Fig. 1:

```
translate(ProjectionTerm,DatabaseGoal,SQLQueryTerm):-
   init_gensym(var),
   init_gensym(rel),
   tokenize_term(DatabaseGoal,TokenDatabaseGoal),
   tokenize_term(ProjectionTerm,TokenProjectionTerm),
   disjunction(TokenDatabaseGoal,Disjunction),
   query_generation(Disjunction,TokenProjectionTerm,SQLQueryTerm),
   printqueries(SQLQueryTerm).
```

In the initialization, counters for the construction of unique identifiers for variables and range variables are initialized to 0. In LPA Mac Prolog `init_gensym/1` is a built-in predicate, in other Prologs it can be implemented as

```
init_gensym(Root):-
   nonvar(Root),
   retract_all('$gensym$'(Root,_)),
   assert('$gensym$'(Root,0)).
```

4.1 Lexical analysis

In the lexical analysis (or *tokenization*) phase each distinct variable in the database goal and the projection term is instantiated with a unique identifier, a term of the form `var(VarId)`, and constant values are wrapped into terms of the form `const(Value)` (Note that because of the instantiation of variables it may be necessary to work with a copy of the original terms).

```
% --- tokenize_term(Term,TokenizedTerm) ----------------------

tokenize_term(var(VarId),var(VarId)):-
   var(VarId),
   % --- uninstantiated variable: instantiate with unique identifier
   gensym(var,VarId).

tokenize_term(var(VarId),var(VarId)):-
   nonvar(VarId).

tokenize_term(Constant,const(Constant)):-
   nonvar(Constant),
   functor(Constant,Constant,0).

tokenize_term(Term,TokenizedTerm):-
   nonvar(Term),
   Term \= var(_),
   Term \= const(_),
   Term =.. [Functor|Arguments],
   Arguments \= [],
   tokenize_arguments(Arguments,TokenArguments),
   TokenizedTerm =.. [Functor|TokenArguments].
```

`tokenize_arguments/2` is called with the arguments of the current term in a list. This predicate calls `tokenize_term/2` for each element of the list until the list of arguments is exhausted.

Example

In the sample query, the projection term and the database goal argument are tokenized to

```
big_planes(const(munich), var(var1), var(var2), var(var3))
```

and

```
var(var0)^
  (flight(var(var0), const(munich), var(var1), var(var2)),
   plane(var(var2), var(var3)),
   var(var3)>avg(var(var4), var(var5)^plane(var(var5), var(var4))))
```

respectively. Every variable is instantiated, e.g. `Dest` to `var(var1)`.

□

4.2 Syntax analysis

In the syntax analysis (or *parsing*) phase, a possibly complex database goal argument is transformed into a list of disjunctive goal conjunctions.

The conjunctions are computed through the predicate `linearize/2` which takes as input the current database goal and returns a right-linear conjunction of goals in its second argument. If the input argument contains a disjunction operator, more than one conjunction will result.

All conjunctions are collected in a list by calling `findall` with `linearize(DatabaseGoal, Conjunction)` as the goal argument:

```
disjunction(Goal,Disjunction):-
   findall(Conjunction,linearize(Goal,Conjunction),Disjunction).
```

The projection term is not subject to a syntax analysis.

Example

The result of the syntax analysis for the database goal of the sample query is a list with the one element

```
[var(var0)^
  (flight(var(var0), const(munich), var(var1), var(var2)),
   plane(var(var2), var(var3)),
   var(var3) > avg(var(var4), var(var5)^plane(var(var5), var(var4))))]
```

□

4.3 Code generation

The code generation requires that the mapping of Prolog predicates and operators to SQL relation tables and operators be defined. With this definition, the disjunctive database goal conjunctions are translated to query terms each representing one SQL query which are then UNIONed.

4.3.1 Mapping function from Prolog to SQL

The schema information of the database to be accessed is represented through the predicates

```
relation(PrologFunctor,Arity,SQLTableName).
attribute(ArgumentPosition,SQLTableName,SQLAttributeName).
```

Example

In the sample database, the schema information is represented by the clauses

```
relation(flight,4,'FLIGHT').
relation(plane,2,'PLANE').

attribute(1,'FLIGHT','FLIGHT_NO').
attribute(2,'FLIGHT','DEPARTURE').
attribute(3,'FLIGHT','DESTINATION').
attribute(4,'FLIGHT','PLANE').
attribute(1,'PLANE','TYPE').
attribute(2,'PLANE','SEATS').
```

□

The mapping of Prolog operators that can be translated to SQL operators is defined through

```
comparison(PrologOperator,SQLOperator).
negated_comparison(PrologOperator,SQLOperator).
aggregate_functor(PrologFunctor,SQLFunctionName).
```

It is assumed that the Prolog arithmetic operators are the same as in SQL. If this were not the case, an additional mapping predicate would have to be defined for arithmetic operators.

4.3.2 Code generation organization

The list of disjunctive database goal conjunctions is traversed recursively until the empty list is reached. Each conjunction is translated to a query term of its own, and the individual query terms are collected in a list.

```
query_generation([],_,[]).

query_generation([Conj|Conjunctions],ProjectionTerm,[Query|Queries]):-
   projection_term_variables(ProjectionTerm,InitDict),
   translate_conjunction(Conj,FROM,WHERE,InitDict,Dict),
   translate_projection(ProjectionTerm,Dict,SELECT),
   Query = query(SELECT,FROM,WHERE),
   query_generation(Conjunctions,ProjectionTerm,Queries).
```

Any variable in the projection term is implicitly existentially quantified by definition. The subgoal `projection_term_variables/2` returns an initial dictionary, which is a list of variable identifiers together with the information that the corresponding variable is existentially quantified. [1]

`translate_conjunction/5` takes as input the current goal conjunction and initial dictionary. This predicate computes the FROM and the WHERE clause of an SQL query. It returns a dictionary which contains for each variable identifier its mapping to a qualified attribute name and its type of quantification.

`translate_projection/3` takes as input the original projection term and the dictionary to return the SELECT clause of an SQL query. Note that wrappers are needed if the projection term consists of more than one variable or constant, but that these wrappers are not considered in the translation.

The query term corresponding to the current conjunction is then assembled by the subgoal `Query = query(SELECT,FROM,WHERE)`, and `query_generation/5` is called recursively with the rest of the disjunctive conjunctions.

4.3.3 Goal translation

In `translate_conjunction/5` a conjunction is broken down into a simple goal and the rest of the conjunction. A simple goal is either a(n)

- simple database goal,
- arithmetic expression,
- comparison operation,
- negated goal conjunction, or
- aggregate function term.

`translate_goal/5` translates simple goals. Two cases must be considered: database goals and arithmetic expressions which may bind variables, and comparisons and negated conjunctions which require all arguments to be bound.

Translation of simple database goals

The first clause of `translate_goal/5` takes as input the current goal and dictionary. It returns items for the FROM and the WHERE clause of the final query, and a new dictionary with the mapping information of the variables bound by the current goal.

[1] Note that for efficiency reasons `projection_term_variables/2` could be evaluated once prior to the generation of queries.

```
translate_goal(SimpleGoal,[From],Where,Dict,NewDict):-
   % --- positive goal binds variables ------
   functor(SimpleGoal,Functor,Arity),
   translate_functor(Functor,Arity,From),
   SimpleGoal =.. [Functor|Arguments],
   translate_arguments(Arguments,From,1,Where,Dict,NewDict).
```

The functor of the current goal is extracted. `translate_functor/3` tests whether this functor can be mapped to an SQL relation table and generates, via `gensym/2`, a new unique identifier, a *range variable* or *alias*, for the current relation table.

```
translate_functor(Functor,Arity,rel(RelationName,Alias)):-
   relation(Functor,Arity,RelationName),
   gensym(rel,Alias).
```

The arguments of the current goal are collected in a list. This list is traversed by `translate_arguments/6` and the mapping of each argument to a qualified attribute is computed by `translate_argument/4` for the three possible cases: the current argument is a constant, it is the first occurrence of a variable, or it is a variable which has occurred previously.

Constant arguments translate to equality comparisons of the attribute corresponding to the current argument position and the constant value.

```
translate_argument(const(C),rel(Table,Alias),Pos,Comp,Dict,Dict):-
   attribute(Pos,Table,Attribute),
   Comp = [comp(att(Alias,Attribute),=,const(C))].
```

If the current argument is a variable which has not occurred previously, then this variable, together with the information that it is universally quantified and the mapping to a qualified attribute, is added to the dictionary.

```
translate_argument(var(VarId),rel(Table,Alias),Pos,[],Dict,NewDict):-
   attribute(Pos,Table,Attribute),
   add_to_dictionary(VarId,Alias,Attribute,all,Dict,NewDict).
```

Otherwise, if the current argument is a variable which has occurred in a previous database goal already `lookup/4` succeeds then an equality expression of the mapping of the first occurrence of the variable and its current mapping is constructed.

```
translate_argument(var(VarId),rel(Table,Alias),Pos,Comp,Dict,Dict):-
   lookup(VarId,Dict,PrevAlias,PrevAtt), attribute(Pos,Table,Attribute),
   Comp = [comp(att(Alias,Attribute),=,att(PrevAlias,PrevAtt).
```

Example

`translate_simple_goal/5` of the first subgoal `?- flight(FNo,munich,Dest,Type)` of the sample query returns

```
FROM = [rel(FLIGHT,rel0)]
WHERE = [comp(att(rel0,DEPARTURE,=,const(munich))]
```

`translate_simple_goal/5` of the second subgoal `?- plane(Type,Seats)` where `Type` has occurred in the previous goal returns

```
FROM = [rel(PLANE,rel1)]
WHERE = [comp(att(rel1,TYPE),=,att(rel0,PLANE))]
```

□

Translation of arithmetic expressions

The second clause of `translate_goal/5` deals with arithmetic expressions.

```
translate\_goal(Result is Expression,[],WHERE,Dict,NewDict):-
   translate_arithmetic_function(Result,Expression,WHERE,Dict,NewDict).
```

The Prolog operator `is/2` may be called to test whether both instantiated arguments evaluate to the same value, or it instantiates the left argument with the value of the evaluation of the right argument. In the second case, the left argument is added to the dictionary, together with the arithmetic expression in the right argument.

`translate_arithmetic_function/5` calls `evaluable_expression/3` for the translation of the arithmetic expression in the right argument of `is/2`. `evaluable_expression/3` does not modify the dictionary, and it returns the arithmetic expression in SQL notation with qualified attributes.

Aggregate function terms are considered to be evaluable expressions.

```
evaluable_expression(AggregateTerm,Dictionary,AggregateQuery):-
   aggregate_function(AggregateTerm,Dictionary,AggregateQuery).
```

An aggregate function term consists of an aggregate functor, an aggregate variable indicating over which attribute the function is to be computed, and a goal conjunction which must have in at least one argument position the aggregate variable.

```
aggregate_function(AggregateTerm,Dict,AggQuery):-
   AggregateTerm =..[AggFunctor,AggVar,AggGoal],
   aggregate_functor(AggFunctor,SQLAgg), conjunction(AggGoal,AggConj),
   aggregate_query_generation(SQLAgg,AggVar,AggConj,Dict,AggQuery).
```

The translation of an aggregate function term is similar to the translation of a regular database goal and projection term with the notable exception being the treatment of the universally quantified variables. `translate_grouping/3` extracts from the current dictionary all variables occurring in the aggregate function term which are not existentially quantified and collects them in an attribute list which becomes the GROUP BY clause of the aggregate function query.

```
aggregate_query_generation(Function,FunctVar,AggGoal,Dict,AggQuery):-
   translate_conjunction(AggGoal,FROM,WHERE,Dict,TmpDict),
   set_difference(TmpDict,Dict,AggDict),
   translate_projection(FunctVar,AggDict,SELECT),
   translate_grouping(FunctVar,AggDict,GROUP),
   AggQuery = agg_query(Function,SELECT,FROM,WHERE,GROUP).
```

Example

The aggregate function term `avg(S,T^plane(T,S))` in the comparison operation translates to the aggregate function subquery term (with empty WHERE and GROUP BY clauses):

```
agg_query('AVG',[att(rel2, 'SEATS')],[rel('PLANE', rel2)],[],[])
```

□

Translation of goals which do not bind variables

Database goals that do not bind variables are negated conjunctions and comparison operations. This is expressed by having the same variable `Dict` in the fourth and fifth argument position of the appropriate clauses of `translate_goal/5`.

Negated database goal conjunctions translate to negated existential subqueries in the WHERE clause of a query, whereas the translation of comparisons simply replaces variables with qualified attributes. Negated comparison operations are rewritten as positive comparisons with the inverse comparison operator.

Negated goals may be conjunctions themselves, and they do not return variable bindings. Hence, if the negated goal is not a comparison operation, `translate_conjunction/5` is called for the translation of the goal within the scope of the negation operator `not/1`, and the dictionary is returned unchanged.

```
translate_goal(not NegatedGoals,[],NegatedSubquery,Dict,Dict):-
   functor(NegatedGoals,Functor,2),
   not comparison(Functor,_),
   translate_conjunction(NegatedGoals,From,Where,Dict,_),
   NegatedSubquery = [negated_existential_subquery([*],From,Where)].
```

If the goal is a comparison operation, then the SQL comparison operator corresponding to the Prolog comparison operator is retrieved, and the left and right arguments of the comparison are translated.

```
translate_goal(not CompGoal,[],CompOp,Dict,Dict):-
   CompGoal =.. [CompOperator,LeftArg,RightArg],
   comparison(CompOperator,SQLOperator),
   negated_comparison(SQLOperator,NegOperator),
   translate_comparison(LeftArg,RightArg,NegOperator,Dict,CompOp).

translate_goal(CompGoal,[],CompOp,Dict,Dict):-
   CompGoal =.. [ComparisonOperator,LeftArg,RightArg],
   comparison(ComparisonOperator,SQLOperator),
   translate_comparison(LeftArg,RightArg,SQLOperator,Dict,CompOp).
```

Arithmetic expressions and aggregate function expressions may both appear as arguments of comparison operations.

Sample query

The evaluation of `code_generation/3` returns the following query term which consist of a single query with a SELECT, FROM, and WHERE clause represented as lists. The WHERE clause consists of a comparison with a constant value, a join condition, and a comparison with the AVG aggregate function subquery with empty conditions and no grouping attributes.

```
[query(
 [const(munich),
  att(rel0, 'DESTINATION'),
  att(rel0, 'PLANE'),
  att(rel1, 'SEATS')],
 [rel('FLIGHT', rel0), rel('PLANE', rel1)],
 [comp(att(rel0, 'DEPARTURE'), =, const(munich)),
   comp(att(rel1, 'TYPE'), =, att(rel0, 'PLANE')),
   comp(att(rel1, 'SEATS'), >, agg_query(
   'AVG',[att(rel2, 'SEATS')],
    [rel('PLANE', rel2)],
    [],
    []))
 ])
]
```

After the translation of the goal conjunction the dictionary contains the following entries:

```
[dict(var0, rel0, 'FLIGHT_NO', existential),
 dict(var1, rel0, 'DESTINATION', existential),
```

```
 dict(var2, rel0, 'PLANE', existential),
 dict(var3, rel1, 'SEATS', existential)])
```

During the translation of the aggregate function term the dictionary contains entries for the aggregate function goal, but these entries are released after this translation has terminated.

4.4 Output

The output predicates print the query term in correct SQL notation to the standard output device. `print_queries/1` calls `print_query/1` to print each query in the list of disjunctive queries. The keyword UNION is printed in between two disjunctive queries, and a semicolon ";" ends a query. For non-empty clauses, `print_clause/3` prints the clause keyword, (i.e. one of SELECT, FROM, WHERE, and GROUP BY), the corresponding column list separated by the appropriate separator, and prints a new line character at the end of each clause. `print_column/2` simply prints one column in the appropriate format to the screen. Some sample clauses are shown here (Note that little care is taken to avoid empty lines):

```
printqueries([Query|Queries]):-
   Queries \= [],
   print_query(Query),
   nl,
   write('UNION'),
   nl,
   printqueries(Queries).

print_query(query(Select,From,Where)):-
   print_clause('SELECT',Select,','),
   nl,
   print_clause('FROM',From,','),
   nl,
   print_clause('WHERE',Where,'AND'),
   nl.
...
print_clause(Keyword,[Column|RestColumns],Separator):-
   write(Keyword),
   write(' '),
   print_clause([Column|RestColumns],Separator).

print_clause([Item,NextItem|RestItems],Separator):-
   print_column(Item),
   write(' '),
   write(Separator),
   write(' '),
   print_clause([NextItem|RestItems],Separator).
...
```

```
print_column('*'):-
   write('*').
   print_column(att(Alias,Attribute)):-
   write(Alias),
   write('.'),
   write(Attribute).
...
```

Example

For the sample query, the output (with empty lines deleted manually) is

```
SELECT munich, rel0.DESTINATION, rel0.PLANE, rel1.SEATS
FROM FLIGHT rel0, PLANE rel1
WHERE rel0.DEPARTURE = munich AND rel1.TYPE = rel0.PLANE AND rel1.SEATS >
 (SELECT AVG(rel2.SEATS)
  FROM PLANE rel2);
```

□

4.5 Problems and restrictions

The compiler described in this report suffers from some deficiencies. The first one is inherent to any compiler which translates fragments of a more expressive language to a less expressive language:

- Although the definition of a database access is formulated entirely in Prolog, SQL shows through in some places.

  For example, complex datastructure cannot be used in database access requests because SQL does not allow data with a complex structure (except strings). Furthermore, for any SQL predicate or function to be available, a suitable mapping from Prolog to SQL has to be defined. Finally, the validity of certain SQL queries can be determined during the database evaluation only, and not at compile time; the GROUP BY clause is an example.

The other deficiencies are restrictions of this particular implementation:

- The compiler does not implement the full SQL query language. A number of arithmetic functions and other predicates are missing. Furthermore, the compiler does not produce HAVING clauses nor ORDER BY clauses. However, these can be added easily:
  - comparisons involving free variables translate to comparisons in the HAVING clause, and
  - the projection term can be used to generate the ORDER BY clause.

- In its present form the compiler does not provide any support for the user. There are no error messages, no help is available, nor is there an indication of the location where an error was found. This is not acceptable for interactive use, but for the compilation of database access requests during the evaluation of an application program these features are of limited value.

Restrictions

The following restrictions hold for the input arguments of the compiler. However, these restrictions do not reduce the expressive power of the database access language.

- The functors `var/1` and `const/1` must not be used as functors of database goals.

  These functors are used to distinguish variable identifiers and constant values in the compiler for an efficient clause selection.

- All projection term variables must be bound through the database goal.

  The projection term determines the SELECT clause which can only contain qualified attributes or constant values. Only instantiated variables can be translated to qualified attributes.

- All explicit variable quantifications (through the `^/2` operator) must precede the database goal.

  Existentially quantified variables must be in the dictionary before the database goal is translated because during the translation variables are considered to be universally quantified if they are not in the dictionary already.

- The goal argument of an aggregate function term can only be a conjunction of database goals.

  The SQL aggregate functions are not defined for a UNION of queries.

- Negation operators must precede comparison operations directly.

  The translation of negated conjunctions of comparisons is not implemented correctly in the current version of the compiler (see section A3. for details).

In SQL queries can in general be expressed in many ways, and it is left to the database query optimizer to find an optimal evaluation of the query. However, two optimizations could be added to the compiler rather easily:

- Disjunctive comparisons translate to ORed comparisons in the WHERE clause instead of UNIONed queries.

  For this, the greatest conjunction common to all disjunctive database goal conjunctions is extracted from the disjunctive database goal and translated. If what is left of the database goal consists of comparisons only, then these comparisons are translated to SQL comparisons and are collected into the WHERE clause joined by the junctor OR.

Prolog	Version	Type	Machine	CPU	Clock (MHz)	Memory (MB)
LPA MacProlog	4.0	Compiler	Macintosh PB 170	68030	25	4
OpenProlog	1.0d36	Interpreter				
Prolog II+	2.3m4	Compiler				
C Prolog	1.5	Interpreter	SUN 3/50	68020	16.7	4
SEPIA	3.0.7	Compiler				
Quintus Prolog	3.0	Compiler	SUN SparcStation 1	Sparc	33	12
			SUN Sparc 4/290			32
Prolog-2	2.36	Interpreter	486 DOS	80486	40	16

Table 1: Prolog environments and machines

- Type information in the database schema representation is exploited for consistency checks during the translation, e.g. type compatibility between a constant value and an attribute type.

  For this, the representation of schema information in the compiler must be extended to contain type information (cf. Appendix D)

5 Benchmarks

5.1 Implementations

The Prolog to SQL compiler was developed under LPA MacProlog on a Macintosh. It has been ported successfully with the full range of features to Prolog-2 on a DOS PC, SEPIA Prolog of ECRC and C-Prolog on a SUN 3/50, Quintus Prolog on a SparcStation and a SUN 4/290, and OpenProlog of Trinity College Dublin and Marseille Prolog II+ on a Macintosh.

The Prolog environments and the machines used are given in table 1.

5.2 Porting problems

The compiler code is about 42 KB long. Great care has been taken to rely as little as possible on features specific to a particular Prolog implementation. Porting the code is easy: any standard Edinburgh Prolog should be able to consult and run the compiler.

Furthermore, the compiler was written according to (my understanding of) R. OKeefe's Prolog programming style guidelines [21]. This has made the code at the same time readable and efficient.

Compiler

For the compiler itself only minor modifications were necessary for the different Prolog environments.

- In the third clause of `tokenize_term/2` the subgoal `functor(Const,Const,0)` tests whether the first argument is in fact a constant with the functor `Constant` and the arity 0. In SEPIA Prolog and Quintus Prolog this subgoal causes a runtime error if it is called with a complex term in its first and second argument. In LPA MacProlog and OpenProlog this call simply fails.

- Quintus Prolog requires `\+/1` to denote negation (other Prologs allow it for compatibility reasons). Hence, all occurrences of `not/1` in the bodies of rules must be replaced through `\+/1`.

- OpenProlog does not have the operator `\=/2`. All calls to `A \= B` have to be replaced by `not (A = B)`.

- The Marseille Prolog II+ requires switching to Edinburgh syntax and an explicit definition of the `^/2` operator.

Auxiliary predicates

A significant difference between the different Prolog environments is their set of built-in predicates. In the compiler, this concerns only auxiliary predicates for counters and benchmarks.

In LPA MacProlog and Quintus Prolog `findall/3`, `gensym/2` and `init_gensym/1` are built-in predicates. In OpenProlog, Prolog-2, C-Prolog, and SEPIA Prolog they had to be implemented manually using `assert/1` and `retract/1` or global variables (through `setval(Id,Value)` and `getval(Id,Value)` respectively).

Another difference between Prolog implementations is the treatment of atoms, numbers, and strings respectively. In standard Prolog, strings are not distinguished from atoms (the built-in predicate `name(Atom,ASCIIList)` returns the ASCII codes of the characters of `Atom` in a list), and numbers are atoms. In some Prologs, e.g. SEPIA, MegaLog, and Prolog-2, strings and numbers are not considered as atoms, but they may be converted to atoms through specialized reformatting predicates. This problem concerns only the implementation of `gensym/2` and `init_gensym/1`.

Benchmarks require access to the system clock. In standard Prolog there exists a predicate `cputime/1` which returns the system time since some defined point in time (system boot, session begin etc.). However, in LPA MacProlog the predicate `ticks/1` with an unbound argument must be used for this, and in OpenProlog `cputime` is an arithmetic function so that `Time is cputime` must be called.

Sample queries benchmarks

The benchmark suite consists of five queries of different complexity. For each query, the code generated by the compiler is also given (with blank lines removed manually).

1. Single relation access:

```
translate(flight(No,Dep,Dest,Type),
 flight(No,Dep,Dest,Type),
SQLQueryTerm).
```

   is translated to the SQL query

```
SELECT rel0.FLIGHT_NO, rel0.DEPARTURE, rel0.DESTINATION,
 rel0.PLANE_TYPE
FROM FLIGHT rel0;
```

2. View access with a selection condition through comparison operations over an attribute and a constant value:

```
translate(capacity(No,Dep,Dest,Type,Seats),
(flight(No,Dep,Dest,Type),
 plane(Type,Seats),
 Type='b-737'),
SQLQueryTerm).
```

   is translated to the SQL query

```
SELECT rel0.FLIGHT_NO, rel0.DEPARTURE, rel0.DESTINATION,
 rel0.PLANE_TYPE, rel1.SEATS
FROM FLIGHT rel0, PLANE rel1
WHERE rel1.TYPE = rel0.PLANE_TYPE AND rel0.PLANE_TYPE = b-737;
```

3. View access with a negated goal:

```
translate(no_planes(No,Dep,Dest,Type),
(flight(No,Dep,Dest,Type),
 not plane(Type,Seats)),
SQLQueryTerm).
```

   is translated to the SQL query

```
SELECT rel0.FLIGHT_NO, rel0.DEPARTURE, rel0.DESTINATION,
 rel0.PLANE_TYPE
FROM FLIGHT rel0
WHERE NOT EXISTS
(SELECT *
 FROM PLANE rel1
 WHERE rel1.TYPE = rel0.PLANE_TYPE);
```

Prolog	Machine	Query 1	Query 2	Query 3	Query 4	Query 5
LPA MacProlog	Macintosh PB 170	67	108	95	65	148
OpenProlog		100	150	138	143	254
Prolog II+		207	333	303	282	500
C-Prolog	SUN 3/50	188	335	293	208	483
SEPIA		30	48	42	23	71
Quintus Prolog	SUN SparcStation 1	13.4	18.3	23.3	15.0	35.0
	SUN Sparc 4/290	6.7	10.0	10.0	6.6	15.0
Prolog-2	486 DOS	3.3	4.3	4.4	3.3	8.7

Table 2: Benchmark results in milliseconds for compilation only

4. Simple aggregate function:

```
translate(X,X is count(S,plane(P,S)),SQLQueryTerm).
```

is translated to the SQL query

```
SELECT COUNT(rel0.SEATS)
FROM PLANE rel0
GROUP BY rel0.TYPE;
```

5. View access with a comparison operation over an aggregate function:

```
translate(big_planes(munich,Dest,Type,Seats),No^
(flight(No,munich,Dest,Type),
 plane(Type,Seats),
 Seats > avg(S, T^plane(T,S))),
SQLQueryTerm).
```

is translated to the SQL query

```
SELECT munich, rel0.DESTINATION, rel0.PLANE_TYPE, rel1.SEATS
FROM FLIGHT rel0, PLANE rel1
WHERE rel0.DEPARTURE = munich AND
 rel1.TYPE = rel0.PLANE_TYPE AND
 rel1.SEATS > (SELECT AVG(rel2.SEATS) FROM PLANE rel2);
```

The predicate `cpu_time(N,Goal,Duration)` to measure the time `Duration` required for `N` evaluations of the goal `Goal` is adapted from R. OKeefeś book, pages 81 - 84 [21]. The benchmark times are given in ms. They are the average execution times of 10 compilations only (table 2)

respectively 10 compilations with output of the SQL query to the screen or a screen-size window in a graphical interface (table 3).

Prolog	Machine	Query 1	Query 2	Query 3	Query 4	Query 5
LPA MacProlog	Macintosh PB 170	373	855	818	358	1248
OpenProlog		390	620	831	570	1077
Prolog II+		327	407	370	290	710
C-Prolog	SUN 3/50	217	375	335	220	561
SEPIA		36	58	60	41	91
Quintus Prolog	SUN SparcStation 1	13.3	26.7	26.7	16.7	40.0
	SUN Sparc 4/290	6.6	13.3	11.7	6.7	21.6
Prolog-2	486 DOS	12.6	15.4	19.8	10.53	8.7

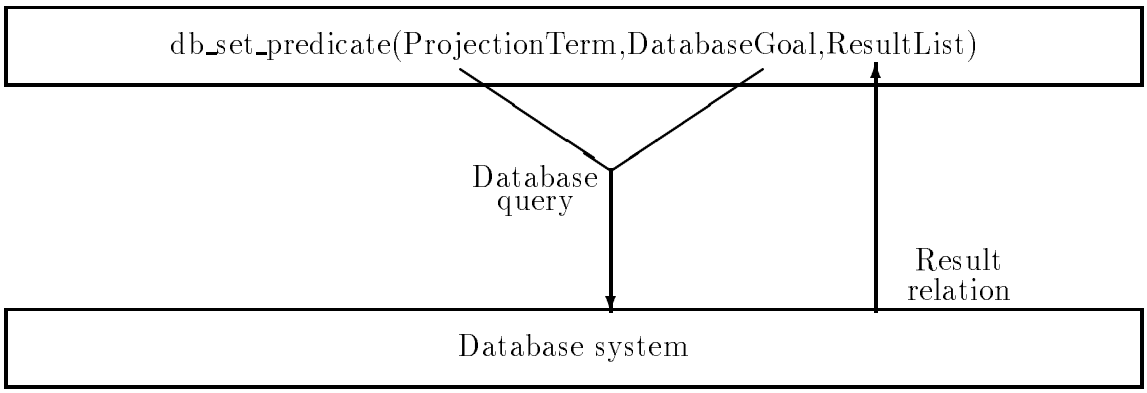
Table 3: Benchmark results in milliseconds for compilation and output to screen

6 Related Work and Outlook

A query generated by the Prolog to SQL compiler is transmitted to an SQL database management system where it is evaluated. In current physically loosely coupled systems, the result relation is made available to the Prolog environment either by asserting it as a whole into the Prolog workspace, or returning one tuple at a time through database cursors or pointers to external buffers. Two major problems are inherent to these systems: First, assertion is expensive both in terms of execution time and of space requirements. Second, view access itself is not sufficiently expressive for higher-order queries ([11] discusses these problems in detail).

6.1 Database Set Predicates

Set predicates, also called *all solutions* predicates, are higher-order constructs which extend the expressive power of Prolog by computing the set (or bag) of solutions of a goal and collect the solutions in a Prolog datastructure [28, 20]. Database set predicates [11] extend the Prolog all solutions predicates with access to external databases. The template and the goal argument of a database set predicate are translated to the appropriate database query, and the result relation is captured in a Prolog list (Fig. 2:).



With database set predicates the higher order features of SQL, namely sorting and aggregate functions (including grouping) can be expressed naturally, resulting in maximally restrictive and highly expressive database queries.

Database set predicates have been implemented in Quintus Prolog and an Oracle database management system under UNIX. Both system components communicate via inter-process communication through sockets which allows the components to be running on different machines in a local area network [4].

6.2 Increasing the expressive power of the database access language

Currently the expressive power of the database access language is restricted to the expressive power of the underlying data manipulation language, i.e. relational algebra with higher-order extensions.

Non-first Normal Form Databases

NF^2 databases [2], which extend relational databases through structured attributes, e.g. tuple-, list-, or set-valued attributes, can also be accessed through database set predicates. Basically, the representation of the schema information in the compiler has to be modified to allow table names as arguments of the `attribute/3` predicate, and in the compilation free variables translate to nesting attributes, and the remaining variables translate to nested attributes.

Example

FLIGHT is an NF^2 relation with the simple attribute *DEPARTURE* and a relation valued attribute *CONNECTION* which itself has the simple attributes *DESTINATION* and *TYPE*.

```
translate(connects_to(Dest,Type), Connection^ (flight(Dep,Connection),
 member(connection(Dest,Type),Connection)),
SQL_NF).
```

would then return the SQL/NF [22] query

```
SELECT DEPARTURE,
 SELECT DESTINATION, TYPE
 FROM CONNECTION
FROM FLIGHT;
```

□

6.3 Datalog

Datalog, a function-free sublanguage of Horn clause logic allows the formulation of recursive queries [25]. Recursive Datalog programs are evaluated bottom-up from the base relation by repeatedly joining the base relation with itself until the fix-point is reached [12]. Generally, a cyclic query evaluation graph is constructed from the Datalog program to control the evaluation of the query (e.g. as in DedGin* [26], LDL [24], or LogiQuel [18]).

Database set predicates can be extended to Datalog queries by considering the projection term as the head of rules, and the goal argument as a disjunction of rule bodies.

Example

The transitive closure of the flight connections could be expressed through the projection term

```
connect(Dep,Dest)
```

and the database goal

```
Transit^(flight(Dep, Dest); flight(Dep, Transit),connect(Transit,Dest))
```

with a `flight/2` database fact.

Such a query could be translated to a Prolog program which repeatedly computes the join over the relations flight and connect and eventually returns as a result the relation `connect`. □

7 Acknowledgments

I thank Martin Emms and Franz Guenthner for their comments on early versions of the report, Hans Leiß for his help with LaTeX, Norbert Fuchs for his Prolog evaluation profiler, and Paul Singleton for reporting errors in the compiler implementation.

References

[1] Quintus Prolog DB Interface User Manual, 1987.

[2] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, April 1984.

[3] J. Bocca. EDUCE – a marriage of convenience: Prolog and a relational DBS. In *Proceedings Third Symposium on Logic Programming*, Salt Lake City, 1986.

[4] B. Britsch. Implementierung einer Datenbank-Mengenschnittstelle in Prolog. Diplomarbeit, Institut für Informatik, Universität Zürich, 1991.

[5] J. Burse. ProQuel: Using Prolog to implement a deductive database system. Technical Report TR 177, Dept. Informatik, ETH Zurich, 1992.

[6] S. Ceri, G. Gottlob, and G. Wiederhold. Interfacing relational databases and Prolog efficiently. In L. Kershberg, editor, *Proceedings First Intl. Conference on Expert Database Systems*, Charleston, 1986. Benjamin-Cummings.

[7] C.L. Chang and A. Walker. PROSQL: A Prolog programming interface with SQL/DS. In L. Kershberg, editor, *Proceedings First Intl. Conference on Expert Database Systems*, Charleston, 1986. Benjamin-Cummings.

[8] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*. Plenum Press, 1978.

[9] M. Danielsson and J. Barklund. Persistent data storage for Prolog. In *Proceedings of DEXA 90*, Vienna, 1990. Springer Verlag.

[10] C. Draxler. Logic programming and databases – an overview over coupled systems and a new approach based on set predicates. Technical Report 90.09, Institut für Informatik, Universität Zürich, 1990.

[11] C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates From Logic Programming Languages*. PhD thesis, University of Zurich, 1991.

[12] F.Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing. In *ACM SIGMOD86*, 1986.

[13] F. Gozzi, M. Lugli, and S. Ceri. An overview of PRIMO: a portable interface to Prolog and relational databases. *Information Systems*, 15(5), 1990.

[14] Y. Ioannides, J. Chen, M. Friedman, and M. Tsangaris. BERMUDA – an architectural perspective on interfacing Prolog to a database machine. In L. Kershberg, editor, *Second Intl. Conference on Expert Database Systems*. Benjamin-Cummings, 1988.

[15] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing Prolog front-end to a relational query system. In *Proceedings SIGMOD*, Boston, 1984.

[16] J. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 2 edition, 1987.

[17] R. Lucas. private communication, 1991.

[18] R. Marti. Research in Deductive Databases at ETH: The LogiQuel project. In S. Spaccapietra, editor, *SI DBTA Proc. Database Research in Switzerland*, 1991.

[19] R. Marti, C. Wieland, and B. Wüthrich. Adding inferencing to a relational database management system. In *Proceedings of BTW 89, Zurich*, Berlin, 1989. Springer Verlag.

[20] L. Naish. *Negation and Control in Prolog*. Springer Verlag, 1986.

[21] R. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[22] M. Roth. *Theory of Non-First Normal Form Relational Databases.* PhD thesis, University of Austin, 1986.

[23] J. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufman, 1988.

[24] S. Tsur. LDL – a technology for the realization of tightly coupled expert database systems. *IEEE Expert*, Fall 1988.

[25] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[26] L. Vieille and A. Lefêbvre. Deductive Database Systems and the DedGin* query evaluator. In *7th British National Conference on Databases*, Edinburgh, 1990.

[27] D.H.D. Warren. Logic programming and compiler writing. *Software Practice and Experience*, 10(11), 1980.

[28] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? *Machine Intelligence*, 10, 1982.

A Evaluation profile

A profile of the translation of the query number 5) may be used to analyze the translation procedure (Table 4). Clearly, for more complex queries it would pay out to organize the dictionary as a balanced tree (e.g. AVL tree) instead of a simple list to speed up dictionary lookup. Furthermore, if_then_else could be used in the tokenization predicates instead of having calls to `var/1` and `nonvar/1` respectively.

A.1 Negated subgoals

The position of the negation operator determines the form of the final SQL query. The three following database access requests evaluate to the same result relation, but are translated to different queries.

```
:- translate(p(T, S), (plane(T, S),not plane('b-737', 200)), Code)

SELECT rel0.TYPE, rel0.SEATS
FROM PLANE rel0
WHERE NOT EXISTS
(SELECT *
 FROM PLANE rel1
 WHERE rel1.TYPE = b-737 AND rel1.SEATS = 200);
```

Goal	call	exit	redo	fail
print_clause/3	6	6	0	0
print_column/1	17	17	0	0
print_clause/2	11	11	0	0
print_clause/4	1	1	0	0
print_query/1	2	2	0	0
printqueries/1	1	1	0	0
query_generation/3	2	2	0	0
projection_arguments/3	7	7	0	0
member/2	37	27	0	10
lookup/4	6	6	0	0
retrieve_argument/3	5	5	0	0
translate_projection/3	2	2	0	0
translate_free_vars/2	1	1	0	0
projection_term_variables/2	2	2	0	0
free_vars/3	1	1	0	0
translate_grouping/3	1	1	0	0
set_difference/3	7	7	0	0
translate_arguments/6	11	11	0	0
add_to_dictionary/6	10	8	0	2
attribute/3	10	10	2	2
translate_argument/6	8	8	0	0
relation/3	9	3	0	6
translate_functor/3	6	3	0	3
translate_goal/5	4	4	0	0
translate_conjunction/5	6	6	0	0
aggregate_query_generation/5	1	1	0	0
disjunction/2	2	2	0	0
conjunction/2	1	1	0	0
aggregate_functor/2	1	1	0	0
aggregate_function/3	2	1	0	1
evaluable_expression/3	2	2	0	0
translate_comparison/5	1	1	0	0
comparison/2	1	1	0	0
projection_list_vars/2	5	5	0	0
tokenize_arguments/2	34	34	0	0
tokenize_term/2	26	26	0	0
translate/3	1	1	0	0

Table 4: Evaluation Profile of query

```
:- translate(p(T, S), (plane(T, S),not (T='b-737'),not (S=200)), Code)

SELECT rel0.TYPE, rel0.SEATS
FROM PLANE rel0
WHERE rel0.TYPE \= b-737 AND rel0.SEATS \= 200;

:- translate(p(T, S), (plane(T, S),not (T='b-737';S=200)), Code)

SELECT rel0.TYPE, rel0.SEATS
FROM PLANE rel0
WHERE rel0.TYPE \= b-737

UNION

SELECT rel1.TYPE, rel1.SEATS
FROM PLANE rel1
WHERE rel1.SEATS \= 200;
```

The reason for these different translations is that negated comparison operations are translated directly into their negative counterpart.

A.2 Incorrect translation of negated comparison conjunctions

Note that it is important that each variable in a negated database goal conjunction occur in a positive database goal. In the following database access request, the variables `T` and `S` are bound by subgoal `plane(T,S)`, but inside the negated goal conjunction they occur only in comparisons. Hence, the negated goal conjunction does not constitute a legal database goal.

The current version of the compiler does not recognize this situation and produces an ill-formed SQL query: there is no FROM clause in the negated subquery.

```
:- translate(p(T,S), (plane(T, S),not (T='b-737',S=200)), Code)
SELECT rel0.TYPE, rel0.SEATS FROM
PLANE rel0 WHERE NOT EXISTS (SELECT *

 WHERE rel0.TYPE = b-737 AND rel0.SEATS = 200);
```

However, such queries can easily be avoided by placing the negation operator directly before a comparison operation as in the examples above.

B New compiler release and new benchmarks

The current version of the Prolog to SQL compiler is v. 1.1. The main improvements are a simple type system, and optional output in the form of Prolog atoms.

B.1 Type system

The Prolog to SQL compiler type system

- correctly quotes string constants in the SQL query output, and
- restricts the use of the arithmetics predicate `is/2` to numerical expressions

For this, the following changes were necessary:

- the Prolog representation of database attribute types now has a type argument

```
% --- attribute(PrologArgPosition,SQLTableName,SQLAttributeName,Type) ----

attribute(1,'FLIGHT','FLIGHT_NO',string).
attribute(2,'FLIGHT','DEPARTURE',string).
...

(1,'PLANE','TYPE',string).
attribute(2,'PLANE','SEATS',integer).
```

- and a rudimentary type system has been included

```
% --- is_type(Type) ------------------------------------------------------

is_type(number).
is_type(integer).
is_type(real).
is_type(string).
is_type(natural).


% --- is_subtype(SubType,SuperType), subtype(SubType,SuperType) -----------
%
% Simple type hierarchy for numeric types
%
% ------------------------------------------------------------------------

is_subtype(integer,number).
is_subtype(real,number).
is_subtype(natural,integer).

subtype(SubType,SuperType):-
   is_subtype(SubType,SuperType).

subtype(SubType,SuperType):-
```

```
   is_subtype(SubType,InterType),
   subtype(InterType,SuperType).


% --- type_compatible(Type1,Type2) ----------------------------------------
%
% Type1 is compatible with Type2 if both are the same types, or if one is
% the subtype of the other
%
% -------------------------------------------------------------------------

type_compatible(Type,Type):-
   is_type(Type).
type_compatible(SubType,Type):-
   subtype(SubType,Type).
type_compatible(Type,SubType):-
   subtype(SubType,Type).


% --- get_type(Constant,Type) ---------------------------------------------
%
% Prolog implementation specific definition of type retrieval
% sepia Prolog version given here
%
% -------------------------------------------------------------------------

get_type('$const$'(Constant),integer):-
   number(Constant).

get_type('$const$'(Constant),string):-
   atom(Constant).
```

- the arithmetic expression clauses were augmented by a type argument

Note that the type system depends on the basic Prolog system type-checking predicates. In Eclipse, the numeric types include `integer`, `number`, and `real`; the type `natural` is a type provided by the Prolog to SQL compiler and is a subtype of `number` and `integer`.

B.2 Atom Output

A second output format is now possible: instead of writing the output to the screen, the SQL query is now put into a Prolog atom. This allows passing on the SQL query to other Prolog predicates, e.g. to the ProDBI interface of Quintus and other Prologs.

Note, however, that this form of output will cause problems if the length of an SQL query is greater than the maximum length of a Prolog atom (which differs from one Prolog system to

Prolog	Machine	Output	1	2	3	4	5	6	7
Eclipse	Sparc 10	yes	4.9	6.7	6.7	4.9	11.6	11.6	18.3
		no	3.3	6.7	4.9	5.0	10	8.3	8.3
LPA MacProlog	PB 180	yes	230	783	750	417	1033	1033	716
		no	65	101	92	65	142	150	150
LPA MacProlog	Mac IIfx	yes	188	385	380	172	636	690	386
		no	30	53	46	32	73	78	78

Table 5: Benchmarks in milliseconds for compilation

the other). [2]

In the benchmark suite, the benchmark number 7 produces a Prolog atom holding the SQL query (manual layout to improve readability):

```
query_atom(
  'SELECT "munich" , rel1.DESTINATION , rel1.PLANE_TYPE , rel2.SEATS
   FROM FLIGHT rel1 , PLANE rel2
   WHERE rel1.DEPARTURE = "munich" AND
         rel2.TYPE = rel1.PLANE_TYPE AND
         rel2.SEATS >
     (SELECT AVG(rel3.SEATS) FROM PLANE rel3 ) ')
```

B.3 New benchmarks

The performance of the Prolog to SQL compiler has been measured for Eclipse (v.3.3.6) [3] on a Sun Sparc 10 workstation with 32 MB, and LPA MacProlog (v.4.5) on a Macintosh PowerBook 180 (33 MHz, 8 MB memory, 2 MB allocated to LPA MacProlog) and a Mac IIfx (40 MHz). The benchmark suite is started through the goal

```
?- benchmark(N,No,D).
```

where `N` is the number of benchmark runs, `No` the number of the benchmark compilation (currently, `No` is one of $1, \ldots, 7$), and `D` the duration in milliseconds.

The times given in table 5 are the average of 10 compilations with and without output to the screen.

[2] The Prolog to SQL compiler is now also being distributed as part of the ProDBI database access software of Keylink Computers Ltd. This software allows accessing a variety of Prolog systems, e.g. Quintus, Sicstus, LPA MacProlog, etc. to commercial relational database systems, e.g. Oracle, Sybase, Ingres, etc.

[3] Eclipse is the name of the Logic Programming environment of the ECRC – SEPIA Prolog is part of this environment.