Interfacing a Natural Language Front-End to a Relational Database

Ioannis Androutsopoulos

M.Sc. Dissertation

Department of Artificial Intelligence

University of Edinburgh

1992

Abstract

Masque is a powerful natural language query interface for databases, developed at the University of Edinburgh. The system accepts written English questions and transforms them into Prolog queries, that are evaluated against the Prolog database.

The purpose of this project was to modify Masque, so that it can be used as a front-end to commercial relational databases, without losing the impressive linguistic coverage, efficiency and portability of the original system.

Several methods for the evaluation of the particular Prolog queries that Masque generates against relational databases were explored, and a modified Masque version, called Masque/SQL, was implemented.

Masque/SQL answers the user's questions by generating suitable SQL queries, that are executed by the relational database management system. The full linguistic coverage of the original Masque is still available, and the new system can be used with any relational database supporting SQL.

Masque/SQL was tested with a world geography knowledge domain, and answered all questions almost as fast as the original Masque version.

Acknowledgements

I would like to thank my two supervisors, Dr G. Ritchie and Dr P. Thanisch, for their guidance throughout my MSc project.

I am also grateful to Mr E. Dee for his continuous help with Ingres, and Dr C. Draxler for providing me a copy of his PhD thesis.

Finally I would like to thank my parents, my sister and Ian, for their encouragement and support.

Table of Contents

List of Figures

Chapter 1

Introduction

The goal of this research project was to attach an existing natural language front-end, called MASQUE, to a commercial relational database, and test the resulting system.

This introductory chapter discusses the motivation for the project, it briefly presents the project's main achievements, and highlights the contents of the following chapters.

1.1 Motivation

Masque is a natural language query interface for databases, developed at the University of Edinburgh. The system, a descendant of Warren & Pereira's CHAT-80 [Warren & Pereira 82], is well-known for its extensive linguistic coverage, its efficiency and its portability.

Masque is currently implemented in SICStus Prolog. It can be easily configured for a variety of knowledge domains, and in most cases it manages to generate answers to complex written English questions in a couple of seconds. However, until this project was carried out the system could only be used as a front-end to Prolog databases: Masque answers each question by transforming it into a Prolog query, which is evaluated against the Prolog database. This method does not allow Masque to be used as a front-end to real-world (usually relational) commercial databases.

It would, therefore, be interesting to explore the possibility of modifying Masque, so that it can be used as an interface to commercial relational databases, without losing the impressive efficiency, linguistic coverage and domain portability of the original system. Such a modification would enable Masque to be used in real-life applications, and it could help demonstrate the feasibility of powerful natural language database interfaces[1].

The goal of this project is also interesting from another point of view: since Masque transforms natural language questions into Prolog queries, attaching it to a relational database requires devising a mechanism to evaluate Prolog queries against relational databases. Such Prolog-to-relational database links are currently an important area of research.

1.2 What has been achieved

The modified version of Masque, called Masque/SQL, that was produced during this project, transforms each user question into an appropriate SQL query. The SQL query is executed against the relational database, and the results are reported back to the user.

Masque/SQL was tested using a world-geography domain, similar to the existing *geog80* sample domain of the original Masque. The system generated appropriate SQL code in all cases, and was able to answer all the sample questions that accompany the *geog80* code, almost as fast as the old system.

Although the new Masque version was tested against a relatively small database, its efficiency should remain unaffected when used with larger databases: each question is transformed into a single SQL query, and the Database Management System (DBMS) of the relational database is left to answer the query using its specialised planning and optimising techniques. In alternative approaches, the DBMS is only

[1]It is not obvious that a natural language front-end is the best interface to a database. See [Bell & Rowe 92] for an interesting comparison of natural language, graphical and artificial language database interfaces.

used to retrieve partial results, and the natural language front-end has the additional responsibility of linking these partial results using some simplistic strategy. The efficiency of front-ends based on such approaches can deteriorate significantly as the size of the database becomes larger.

The full linguistic coverage of the original Masque is still available in the new version, and the new system can be configured for a new knowledge domain as easy as the old one. Masque/SQL has been tested using Ingres v.6.3, but it could be easily ported to any relational database that supports C with embedded SQL.

1.3 Overview of the following chapters

Chapter 2: An Overview of Masque

Masque uses several transformations that turn each English question into a (Prolog) *logical query*. Chapter 2 explains the purpose of these transformations and describes the architecture of the old Masque version.

There are generally two ways to attach Masque to a commercial relational database: (i) extending the Prolog interpreter, so that the external relational database can be treated as part of the Prolog database, or (ii) translating each Masque logical query into a database query, expressed in some query language of the relational database (e.g. SQL). Chapter 2 shows how the Masque architecture would need to be modified in each case.

Chapter 3: The Logical Query Language of Masque

Each logical query expresses precisely what Masque understands to be the meaning of the corresponding English question. The user's question is answered by *evaluating* the logical query against the database (Prolog or relational).

Although the Masque logical queries are actually Prolog expressions, they are not arbitrary Prolog code. There are only a few possible logical query forms, and only certain types of predicates and terms may appear within each logical query. In addition, the semantics of the logical queries are not necessarily as strong as the semantics of Prolog. Thus, the logical queries of Masque can be viewed as expressions of a logical query language, called LQL, which is a subset of Prolog.

Understanding the exact structure and the semantics of the logical queries generated by Masque is of key importance, in order to design a method to evaluate them against the relational database. However, until this project was carried out LQL had never been formally specified. So, the first and necessary step of this project was to provide a formal account of the LQL syntax and semantics, which is presented in chapter 3.

Chapter 4: Evaluating Logical Queries Against a Relational Database

Although the logical queries generated by Masque are actually Prolog expressions, there is no need to evaluate them by executing them as Prolog programs.

Chapter 4 describes alternative methods that can be used to evaluate logical queries against relational databases. Several approaches are discussed, and the techniques used by two existing natural language front-ends to relational databases, the interface developed at the University of Essex and IBM's LanguageAccess, are briefly presented.

Chapter 5: Evaluating LQL Queries by Translating them into SQL

Masque/SQL answers each question by generating an LQL query, which is then translated into SQL. Chapter 5 describes the algorithm used in Masque/SQL to transform LQL queries into SQL code, and explains how each LQL query is evaluated by executing the corresponding SQL code.

Chapter 6: Implementing and Testing Masque/SQL

Chapter 6 describes the architecture and functionality of the new Masque version, focusing on the software modules created or modified during this project. The results of the tests made with the new system are also presented and commented.

Chapter 7: Conclusions and Further Improvements

Chapter 7 summarises the conclusions of this project, and proposes further improvements to Masque/SQL.

Chapter 2

An Overview of Masque

2.1 Introduction

MASQUE (Modular Answering System for QUeries in English) is a natural language query interface for databases. Masque accepts written English questions, referring to a certain knowledge domain (e.g. geography, airplane departures/arrivals etc). Each English question is transformed into a suitable database query, which is executed against the database to retrieve the requested information. The results of the query are then reported back to the user.

For example, given the question '*what is the capital of each European country bordering the Mediterranean?*', Masque would report:

```
Spain Madrid, France Paris, Italy Rome, Greece Athens...
```

Masque is a descendant of CHAT-80, a system created by Warren and Pereira in the early eighties [Warren & Pereira 82]. CHAT-80 was designed as a general and portable natural language interface to an arbitrary database. Although its ability to cope with non-trivial English questions and its efficiency were impressive, it proved to be in many ways idiosyncratic and hard to port between databases and knowledge domains. Since then, several efforts have been made at the University of Edinburgh, to redesign the system, so that it is genuinely portable and efficient. Masque is the outcome of these efforts.

Masque can process quite complex English questions. In most cases the answers are generated in well under five seconds, and if no suitable answer can be found, a 'cooperative' message is generated, that helps the user understand why the system failed to answer the question. Masque also incorporates a domain-editor, which helps configure the system for use in new knowledge domains.

Until this project was carried out, Masque could only be used as a front-end to Prolog databases. The purpose of this project was to modify Masque, so that it can be used as a front end to commercial relational databases. Throughout this dissertation, I will call Masque/PRO the old version of Masque, the one that could only be used as a front-end to Prolog databases. The modified version of Masque, that can be used to query relational databases, will be called Masque/SQL.

This chapter provides a brief overview of the functionality and the architecture of Masque. The intention is to provide enough information, so that the goals of the project and the modifications made to Masque can be understood. Refer to [Auxerre & Inder 86], [Auxerre 86], [Lindop 86], [Spenceley 89], [Sentance 89], [Iliopoulos 92] for a more detailed description of Masque, and some of the improvements that have been made to it by other researchers.

- Masque answers the user's questions by applying several transformations, that turn each English question into a logical query. Section 2 describes these transformations and provides an overview of the Masque/PRO architecture.

- As already mentioned, Masque/PRO could only be used to interface to Prolog databases. Section 3 presents two possible modified Masque architectures, that would enable the system to be used as a front-end to external commercial databases.

2.2 The Masque/PRO architecture

Figure 2–1 shows the main modules of Masque/PRO.

The English question is first analysed syntactically using an *Extraposition Grammar* parser. Extraposition Grammars [Pereira 81] are an enhanced version of Definite Clause Grammars [Pereira & Warren 80], that can be used to express more naturally the syntax of *wh*-questions (*Who/What ... ?*).

The resulting *syntax structure* (a parse tree) is converted by the *semantic interpreter* into a *semantics structure*, roughly a logical expression with unresolved

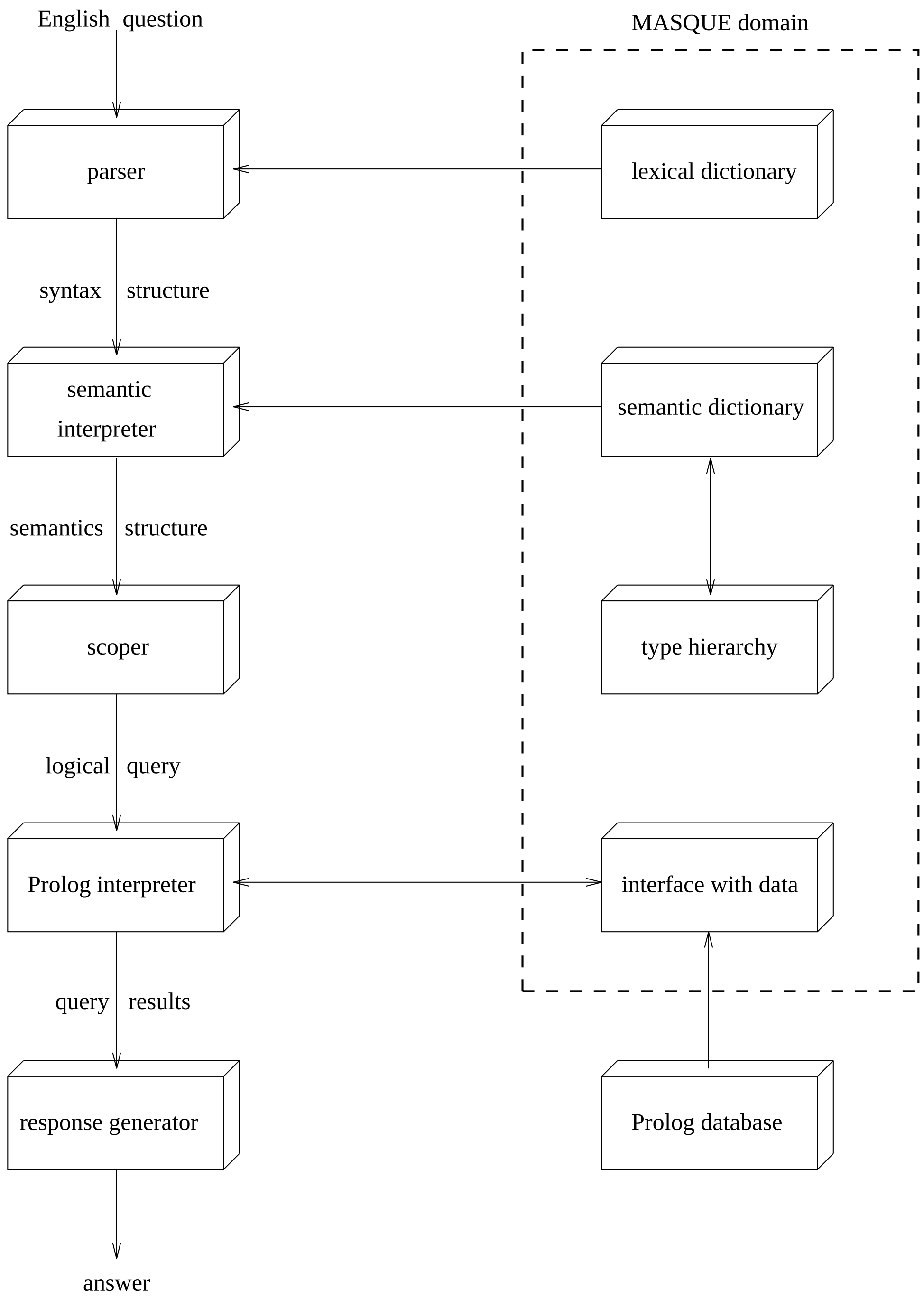

Figure 2–1: The Masque/PRO architecture

quantifier scopes. A *scoper* is then used to convert the semantics structure into a full *logical query*.

Each logical query expresses precisely what Masque understands to be the meaning of the corresponding English question. The logical queries of Masque are expressed using a *logical query language* called LQL (see chapter 3), which is actually a subset of Prolog[1].

Masque/PRO contains several more modules not shown in the diagram, most notably a *logical query optimiser* [Auxerre & Inder 86], [Auxerre 86], [Warren 81]. The modules not shown are not directly relevant to this project.

From the modules shown in figure 2–1, the *lexical dictionary*, the *semantic dictionary*, the *type hierarchy* and the *interface with data* are domain-dependent and have to be created by the *knowledge engineer*, whenever Masque is configured for a new knowledge domain. Masque provides a domain editor, a tool that helps the knowledge engineer create easily the two dictionaries and the type hierarchy.

the lexical dictionary: Although the Extraposition Grammar used by the syntax parser is domain-independent and hard-wired into the Masque code, the morphological information used by the Extraposition Grammar is domain-dependent and has to be defined by the knowledge engineer in the *lexical dictionary*.

For every noun, adjective, verb and preposition expected to appear in the user's questions, a suitable entry must be included in the *lexical dictionary*, showing the syntactic category (e.g. verb, noun) of the word, and the various forms with which the word may appear in a question (e.g. 3rd person form, past tense, participles for a verb, and plural form for a noun).

[1] In fact, Masque first generates expressions of a logical language called DCW (Definite Closed World clauses) [Pereira 82], and a separate module transforms these expressions into LQL. However, the DCW expressions are so close to Prolog, as we know it today, that this intermediate transformation is not discussed here.

Determiners, interrogative pronouns, auxiliary verbs and some other common words are built-in and therefore need not be described in the lexical dictionary [Auxerre & Inder 86].

the semantic dictionary and the type hierarchy: The meaning of each domain dependent word, i.e. of each word mentioned in the lexical dictionary, has to be defined in terms of a logic predicate.

For example, the meaning of the word '*city*' could be described by the predicate $is\_city(X)$, denoting that a question like *'Is Edinburgh a city?'* should be translated into logic as $is\_city(edinburgh)$.

Similarly, the meaning of '*population*' could be expressed by the predicate $population\_of(X, Y)$ (as in $population\_of(athens, 4000000)$), and the meaning of the verb '*borders*' could be expressed by the predicate $borders(X, Y)$ (as in $borders(usa, mexico)$).

In that sense, each domain dependent word introduces a *domain dependent logic predicate.* Each syntax category introduces domain dependent predicates of a particular pattern. The exact form of the logical predicates introduced by each syntax category is described in chapter 3 and, in more detail, in [Auxerre & Inder 86].

Each argument position of a domain dependent predicate corresponds to a particular semantic type. Semantic types are similar to sorts in many-sorted logic. In the examples above, *is_city* would have been declared in the semantic dictionary as *is_city(city_type)*, denoting that the argument of *is_city* must be of type *city_type*. Similarly, *borders* would have been declared as $borders(country\_type, country\_type)$, and $population\_of$ as $population\_of(place\_type, integer\_type)$.

In Masque, the semantic types are organised in a type hierarchy, a forest of trees, where each ancestor type subsumes its descendant types. An example of a type hierarchy is given in figure 2–2.

Masque uses the type hierarchy to check the consistency of the user's questions. For example, it could deduce that the question'*does 5 border Italy?*' is inconsistent: the translation into logic of the question would contain the

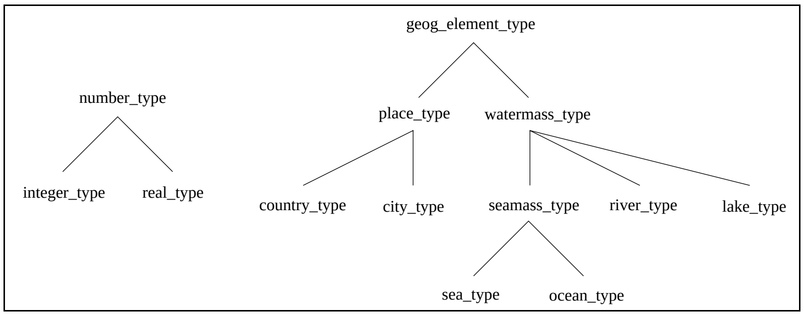

Figure 2–2: An example of a Masque type hierarchy

predicate instance $borders(5, italy)$. However, 5 is of type *number_type*, *number_type* is not subsumed by *country_type*, and the semantic dictionary describes the meaning of *'borders'* as $borders(country\_type, country\_type)$. Hence, the arguments of $borders(5, italy)$ are of inappropriate types, which shows that the question is inconsistent.

On the contrary, the question '*what is the population of Rome?*' is consistent, because the meaning of '*population*' is $population\_of(place\_type, integer)$, and '*Rome*' is of type *city_type*, and therefore of type *place_type*.

the interface with data: Masque uses the domain dependent predicate declarations of the semantic dictionary to generate LQL (Prolog) queries. For example, the question '*what is the population of each city?*' would be translated into the logical query:

```
answer([_10, _20]):- is_city(_10),
                      population_of(_10, _20)
```

where `is_city` and `population_of` derive from the corresponding entries for '*city*' and '*population*' in the semantic dictionary.

Since in Masque/PRO the logical queries are executed directly by the Prolog interpreter, every domain dependent predicate has to be implemented as a Prolog predicate. The Prolog implementations of the domain dependent predicates are placed in the *interface with data*.

Returning to the previous examples, the predicates `population_of` and `is_city` could have been defined as:

```
is_city(C):- city_info(C,_,_).
population(Place, Pop):- country_info(Place, Pop, _);
                         city_info(Place, Pop, _).
```

where the Prolog database is assumed to contain the predicates:

```
country_info(Country_name, Population, Capital)
city_info(City_name, Population, Country)
```

2.3 Alternative Masque architectures

Clearly, the Masque/PRO architecture doesn't allow Masque to be used as a front-end to commercial relational databases, since the logical queries are evaluated by the Prolog interpreter against the Prolog database. There are several ways to modify Masque, so that an external relational database can be used. Most of these fall into two categories:

- Providing an interface that enables the Prolog interpreter to use the external relational database as part of the Prolog database.

- Providing a module that will translate each logical query into a database query, expressed in some query language of the external relational database (e.g. SQL). This is the approach used in Masque/SQL.

Figure 2–3 shows a possible Masque architecture for the first approach. The Prolog interpreter is augmented by an interface, that allows Prolog to use data stored in the external database. Depending on the transparency of the Prolog to DBMS (Database Management System) interface, the *interface with data* may be the same as in the Masque/PRO architecture, or contain more information showing how to connect relations of the external database to Prolog predicates. All other Masque modules remain as in the Masque/PRO architecture. Note that the Prolog database is still available to complement the data stored in the external relational database.

Figure 2–4 shows a possible Masque architecture for the second approach. In this case, the *interface with data* contains information that guides the translator

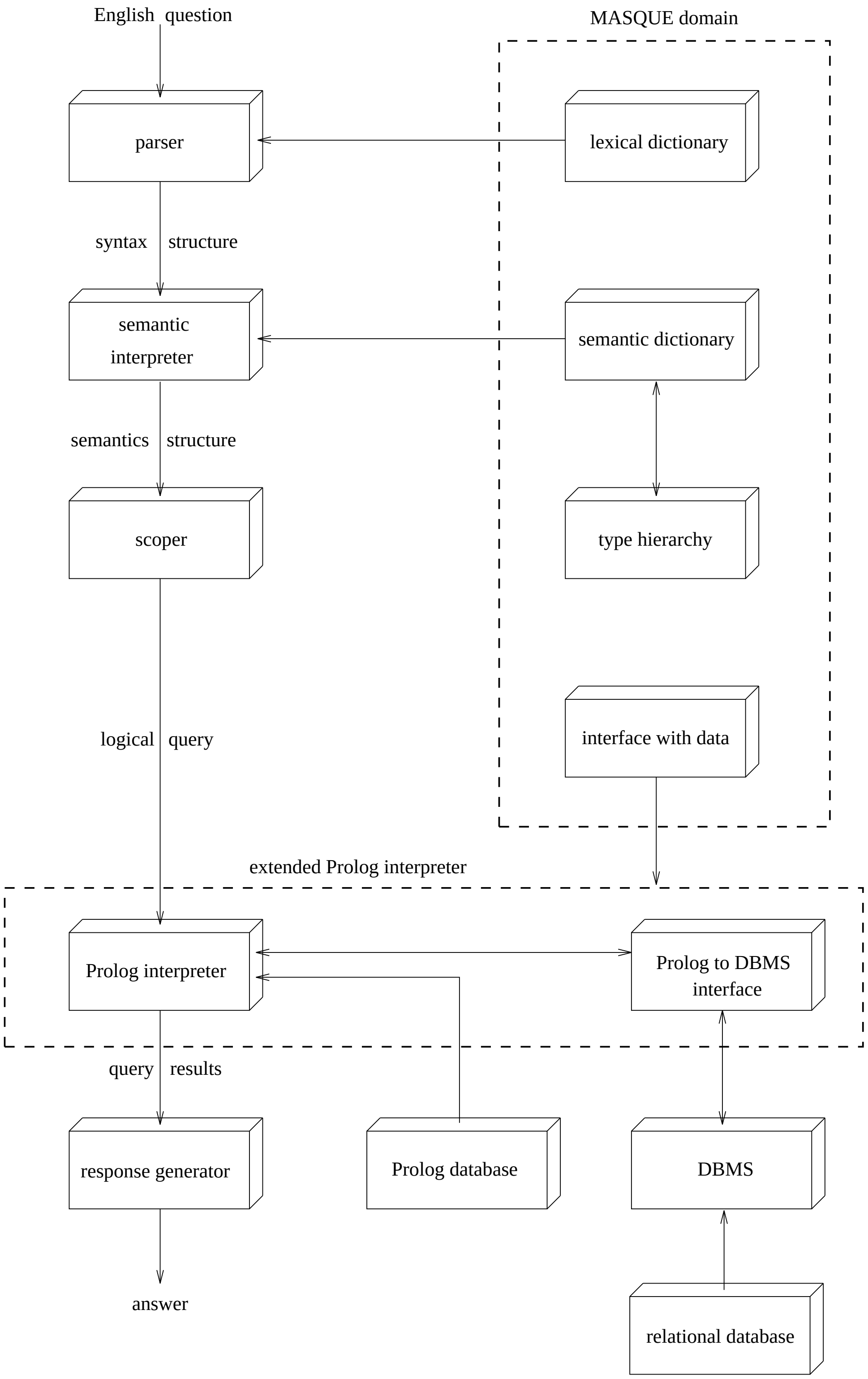

Figure 2–3: Treating the relational database as an extension of the Prolog database

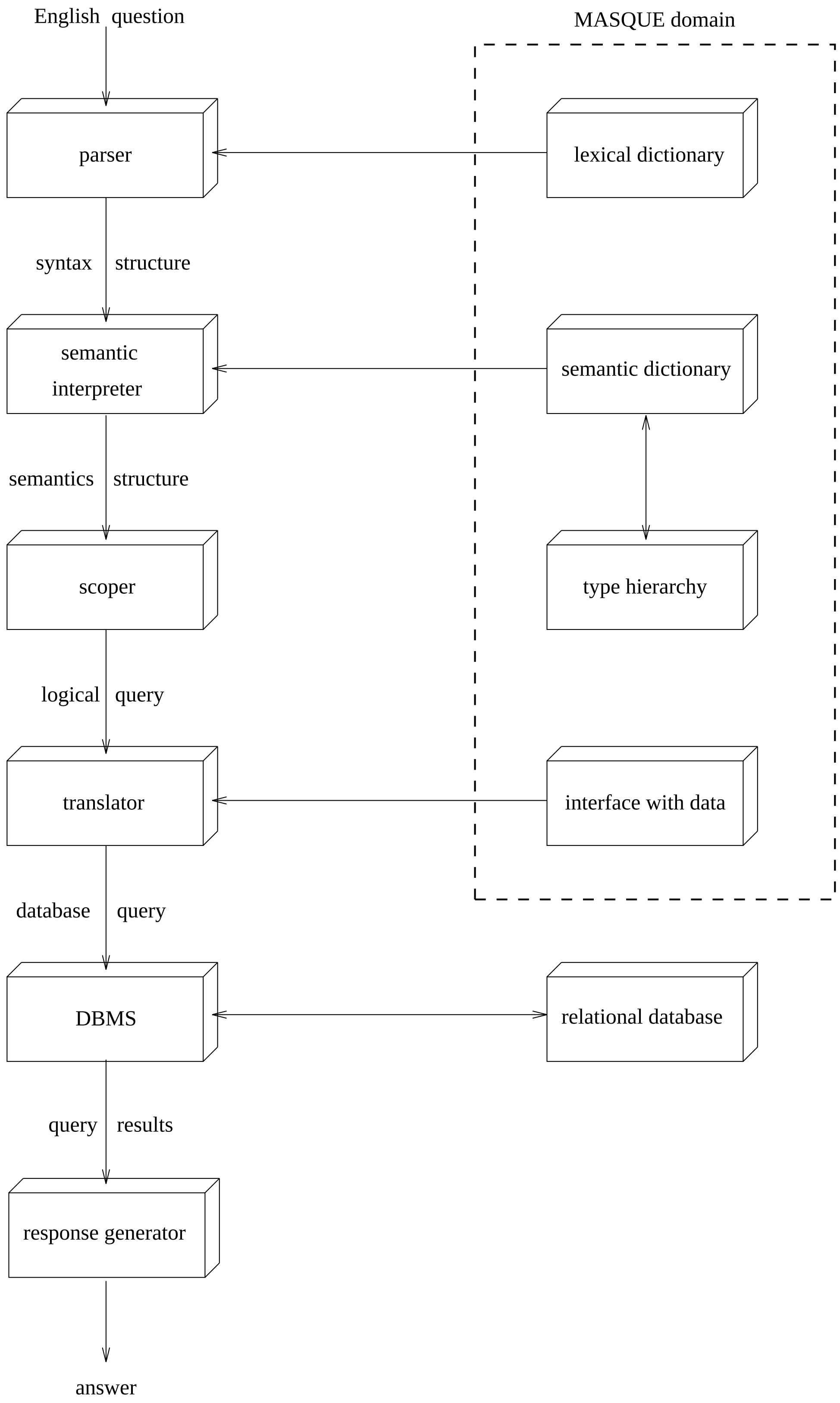

Figure 2–4: Translating each logical query into a database query, expressed in some query language of the relational database

to convert domain dependent predicate instances into appropriate expressions of the database query language. Note that the logical query is no longer executed by the Prolog interpreter.

Both approaches will be discussed in the following chapters.

2.4 Summary

This chapter has provided a brief overview of the functionality and the architecture of Masque/PRO, the old version of Masque. Using the old Masque architecture, only Prolog databases could be accessed. There are generally two methods to modify Masque, so that it can be used to access external relational databases: (a) extending the Prolog interpreter, so that it can access data stored in the relational database, and (b) translating each logical query into a database query, expressed in some query language of the relational database.

Chapter 3

The Logical Query Language of Masque

3.1 Introduction

Masque translates English questions into logical queries, expressions of a logical language used internally by the system. I will call this internal language *the Logical Query Language of Masque* – LQL. Each LQL logical query specifies precisely what Masque understands to be the meaning of the corresponding English question.

LQL is actually a subset of Prolog. In Masque/PRO, the old version of Masque, the logical query is executed directly by the Prolog interpreter, to retrieve the requested information. However, for the purposes of this project, LQL will be viewed as *meaning representation language*[1], used to provide a formal account of the semantics of the user's questions.

---

[1]In [Fernandes *et al* 91] the writers provide a formal description of a meaning representation language, suitable for natural language query interfaces to databases. Although their proposed meaning representation language has little in common with LQL, the style and methodology of this chapter have been greatly influenced by [Fernandes *et al* 91]. See also [Fernandes 90] for a detailed discussion of meaning representation languages for natural language front-ends.

At the time this project was carried out, LQL had never been formally specified; the exact form of the logical queries generated by Masque was often hard to predict, and the meaning of the LQL queries was easily misunderstood.

The purpose of this chapter is to provide a detailed description of the syntax and the semantics of LQL. Such a description is a necessary step, in order to devise a mechanism for evaluating LQL queries against relational databases (see later chapters).

- LQL can be used to express questions about certain kinds of worlds. The next section of this chapter describes the assumed structure of these worlds.

- The third section contains an informal description of the LQL syntax. A context-free grammar description of the LQL syntax can be found in Appendix A.

- The fourth section defines the semantics of LQL.

- The fifth section discusses some cases where the logical queries generated by Masque might not follow the LQL specification of this chapter.

- Finally, the sixth section summarises the chapter.

Examples of LQL logical queries can be found in Appendix C.

3.2 The LQL ontology

LQL can be used to express questions about a world, a *domain* according to the Masque terminology, consisting of entities, entity sets and certain kinds of relations involving entities.

An *entity* is any single element of the world, to which reference can be made. Examples of entities are:

flight *sm712*, the *Pacific Ocean*, *Athens*, the number *24*, the colour *red*

All worlds are assumed to contain the entities nil, $true$, $false$, and the real numbers.

An *entity-set* is a set containing entities. Entity-sets can be finite or non-finite. All standard set-theory notions apply. Examples of entity sets are:

$$\{Spain, France, Italy\}, \quad \{1, 5, 12\}$$

All worlds are assumed to contain the set of real numbers $\mathcal{R}$.

An *entity-pair* is an ordered pair of two world entities, of which the first is a real number. E.g: $(24, airplane)$, $(4.5, car)$. In Masque, entity-pairs are used to express some measurable property of an entity.

An *entity-pair-set* is a set containing entity-pairs. Entity-pair-sets can be finite or non-finite. E.g: $\{(3, car),\ (4, airplane),\ (6.8, bus)\}$

LQL expressions may refer to two kinds of world *relations*:

entity-relations: R is an entity-relation over $set_1 \times set_2 \times \ldots \times set_n$ iff

$$R \subseteq set_1 \times set_2 \times \ldots \times set_n$$

where set_1, set_2, $\ldots$, set_n are entity sets.

e.g: for an entity-relation $city\_in$ we could have:

$$city\_in \subseteq set\_of\_cities \times (set\_of\_countries \cup set\_of\_continents)$$

and $(Athens, Greece)$, $(Athens, Europe)$, $(Edinburgh, Europe)$ could be elements of $city\_in$.

entity-pair-set-to-entity relations: R is an entity-pair-set-to-entity relation (EPSE relation) over

$$entity\_set \times \mathcal{P}(entity\_pair\_set)$$

iff:

$$R \subseteq entity\_set \times \mathcal{P}(entity\_pair\_set)$$

where $\mathcal{P}$ denotes the powerset (i.e. the set of all subsets). Intuitively, an EPSE relation links entity-pair-sets to entities, while an entity-relation links entities to entities.

e.g: we could have an EPSE relation $heaviest\_persons$ defined over:

$$persons \times \mathcal{P}(person\_weights)$$

where

$$person\_weights = \{(80, person_1),\ (75, person_2),\ (80, person_3)\}$$

and $persons = \{person_1, person_2, person_3\}$,

that would 'find' the heaviest persons, i.e:

$$(person_1,\ \ \{(80, person_1),\ (80, person_3)\})$$

and

$$(person_3,\ \ \{(80, person_1),\ (75, person_2),\ (80, person_3)\})$$

would both belong to $heaviest\_persons$, but

$$(person_2,\ \ \{(75, person_2),\ (80, person_3)\}$$

would not.

3.3 The syntax of LQL

This section describes informally the syntax of LQL. A context-free grammar description of the LQL syntax can be found in Appendix A. As the various elements of the LQL syntax will be introduced, some hints about their semantics will also be given. A detailed description of the LQL semantics is given in section 3.4.

3.3.1 Atoms

LQL atoms are used to represent world entities. There are two kinds of atoms in LQL: *numbers* and *symbols*.

Numbers may be unsigned or signed, integers or reals. E.g:

5, 0, -12, 54, $+72$, 3.14, -8.42, $+0.2$

Symbols have the form of character strings. The first character of the string must be a lower case letter; the rest of the string may contain any letter or digit, and possibly the character '_'.

e.g. $flight52$, $asian\_river$, $airplaneBA51$

3.3.2 Pairs

LQL pairs are used to represent world entity-pairs. Pairs have the form *number* : *atom*. E.g: $50\!:\!sm712$, $-3012.7\!:\!car$, $5\!:\!7.2$.

Pairs may also have the form *number*:[*atom*]. The later form is only defined because it appears in some of the logical queries generated by Masque. It has no special meaning and it behaves exactly as the first form.

3.3.3 Sets

LQL sets are used to represent entity-sets and entity-pair sets. LQL sets generally have the form [$element_1$, $element_2$, ..., $element_n$]. There are two types of sets: *atom-sets* (e.g. $[oa512, edinburgh, ba102]$) and *pair-sets* (e.g. $[800 : china, 10 : greece]$).

An LQL set may also have the form [] (empty set).

3.3.4 Variables

Variables have the form _*num*, where *num* is an unsigned integer. Examples of variables are: _17512, _0.

Variables may appear in any place where an atom, a pair or a set can appear, unless otherwise mentioned.

3.3.5 Predicate instances

LQL predicates are used to represent facts that must be true in the world. Formally an LQL *predicate* is an ordered pair (*functor*,*arity*), where *functor* is a symbol containing no upper-case letters, and *arity* is an unsigned integer.

A *predicate instance* of a predicate (*functor*,n) generally (see section 3.3.9 for some exceptions) has the form:

$$functor(arg_1, arg_2, \ldots, arg_n)$$

The exact form of the arguments depends on the predicate. Predicates can be thought as families of predicate instances. Examples of predicate instances are:

$capital\_of(scotland, edinburgh)$, $population(china, 800)$, $continent(\_10)$

There are two kinds of predicates in LQL: *domain-dependent* predicates and *built-in* predicates. These two kinds of predicates will be presented in sections 3.3.8 and 3.3.9.

3.3.6 Conjunction lists

LQL conjunction lists are used to represent lists containing facts, where all facts must be true in the world. A conjunction list has the form $P_1, P_2, \ldots, P_n$, where P_1, P_2, …, P_n are predicate instances or conjunction lists or quantified conjunction lists (see section 3.3.7).

An example of a conjunction list is:

```
country(_17),
population_of(_17, _21),
population_of(china, _22),
greater_than(_21, _22)
```

3.3.7 Quantified conjunction lists

A quantified conjunction list has the form (*List*) or the form:

$$var_1\hat{}\,var_2\hat{}\ldots\hat{}\,var_n\hat{}(List)$$

where *List* is a conjunction list. In the second case var_1, $var_2, \ldots, var_n$ are called *quantified variables*. *List* is called the *body* of the quantified conjunction list.

The meaning of a quantified conjunction list will be defined to be the same as the meaning of its body. Quantified variables are only introduced to ensure the appropriate behaviour of the Prolog predicate `setof`, which is used in Masque/PRO.

An example of a quantified conjunction list is:

```
(_1234^_23^(airplane(_1234),
            _57^(links(_1234,_57,_23),
                 in(europe,_57)
                )
           )
)
```

3.3.8 Domain-dependent predicates

The predicates that appear in an LQL expression are not arbitrary; they are either introduced at the semantic dictionary of a Masque domain (see section 2.2), or they are *built-in* predicates.

This section describes the predicates that are introduced at the semantic dictionary to describe the meaning of domain-dependent words. For each category of words, a pattern of the possible corresponding predicates is given. The reader will recall that variables may appear in any place where an atom, a pair, or a variable can appear, unless otherwise mentioned.

The functor of a domain-dependent predicate instance cannot be the same as the functor of any built-in predicate.

Nouns with no complements introduce predicates of arity 1.

pattern: $functor(atom)$

E.g: the noun *city* could introduce a predicate $city(arg)$, denoting that arg is a city.

Nouns with entity complements: For example, in '*the population of France*', '*of France*' is a complement of the noun '*population*'.

Nouns with entity complements introduce predicates of arity equal to the number of complements plus 1. The first argument corresponds to the entity the noun describes, and the rest of the arguments correspond to the complements.

pattern: $functor(atom, atom, \ldots)$

E.g: the noun *population* could introduce a predicate $population(arg_1, arg_2)$ denoting that arg_1 is the population of arg_2.

Nouns and adjectives acting over set-complements: For example, in '*the average of the areas of the european and american countries*', the noun '*average*' acts over the set containing the areas of the european and american countries. Also, in '*the average population of the european countries*', the adjective '*average*' acts over the set containing the populations of european countries.

Nouns and adjectives acting over set-complements introduce predicates of arity 2. The first argument corresponds to the entity the noun or adjective describes. The second argument corresponds to the set the noun or adjective acts over. The second argument is always a variable, standing for an entity-pair-set.

pattern: $functor(atom, variable)$

E.g: the noun/adjective *average* could introduce a predicate $avg(arg_1, arg_2)$ denoting that arg_1 is the average of the first parts of the pairs in the entity-pair-set arg_2.

Prepositions: introduce predicates of arity 2.

pattern: $functor(atom, atom)$

E.g: the preposition *in* could introduce a predicate $in(arg_1, arg_2)$ denoting that arg_1 is in arg_2.

Note that it is not possible to use prepositions the meaning of which can only be described by predicates of arity greater than 2. For example, one might want to use a predicate $between(arg_1, arg_2, arg_3)$ to describe the meaning of the preposition '*between*' (arg_1 is between arg_2 and arg_3).

Qualitative adjectives: Qualitative adjectives assign a property to a noun, restricting its meaning. For example, in '*european country*', '*european*' restricts the meaning of '*country*'.

Qualitative adjectives introduce predicates of arity 1.

pattern: $functor(atom)$

E.g: The adjective *european* could introduce a predicate $european(arg)$ denoting that arg is european.

Quantitative adjectives: Quantitative adjectives refer to measurable properties of nouns. For example, in '*How large is Scotland?*', '*large*' refers to the area of '*Scotland*'.

Quantitative adjectives introduce predicates of arity 2. The first argument corresponds to the entity the adjective is specifying. The second argument corresponds to the value the adjective is assigning to the specified entity.

pattern: $functor(atom, number)$

E.g: The adjective *large* could introduce a predicate $has\_size(arg_1, arg_2)$, denoting that the size of arg_1 is arg_2.

Verbs introduce predicates with one or more arguments. The first argument always corresponds to the subject of the verb. The other arguments correspond (by the order mentioned) to the direct object (if any), the indirect object (if any) and the complements (if any).

pattern: $functor(atom, atom, \ldots)$

E.g: The verb *borders* could introduce a predicate $bord(arg_1, arg_2)$, denoting that arg_1 borders arg_2.

Domain-dependent predicates are divided into two categories:

type-A predicates: Predicates introduced by nouns with no complements or nouns with entity complements, qualitative adjectives, quantitative adjectives, prepositions, and verbs. Intuitively, type-A predicates correspond to entity-relations.

type-B predicates: Predicates introduced by nouns or adjectives acting over entity-sets. Intuitively, they correspond to EPSE relations, i.e. they provide links between entity-sets and entities.

3.3.9 Built-in predicates

The following predicates are built-in, and they are always available, whether they are mentioned in the semantic dictionary or not. The reader will recall that variables may appear in any place where an atom, a pair, or a variable can appear, unless otherwise mentioned.

length: $length(variable, unsigned\_integer)$ The $variable$ corresponds to an LQL set. This predicate is used to find the cardinality of a set. $variable$ must also appear as a third argument of a $setof$ instance, conjoined to and preceding the $length$ instance. Intuitively, $variable$ must already be bound to a set, when we reach the $length$ instance.

min: $min(atom, variable)$ The $variable$ corresponds to an LQL pair-set. $variable$ must also appear as a third argument of a $setof$ instance, conjoined to and preceding the min instance

max: Same argument patterns as min.

tot: $tot(number, variable)$ The variable corresponds to an LQL pair-set. This predicate is used to add the first parts of the pairs in an LQL pair-set. $variable$ must also appear as a third argument of a $setof$ instance, conjoined to and preceding the tot instance

av: Same argument patterns as tot. This predicate is used to compute the average of the first parts of the pairs in an LQL pair-set.

greater_than: $greater\_than(variable, variable)$ The variables correspond to LQL numbers. Each $variable$ must also appear as an argument of a predicate instance, conjoined to and preceding the $greater\_than$ instance, excluding $setof$ arguments, first $length$ arguments, second min, max, tot, av arguments, other $greater\_than$ and $less\_than$ arguments, and $<$ or $>$ arguments.

Intuitively, the *variable*s must be already bound to numbers, when we reach the *greater_than* instance.

less_than: Same argument pattern as *greater_than*.

true: This is the only LQL predicate with no arguments.

is: $is(variable, variable)$ This predicate is used to unify, in the Prolog sense, two LQL variables. In Masque/PRO '*is*' is not built-in. Masque/SQL has '*is*' built-in for convenience, and because it is very difficult to define it as a domain-dependent predicate (see chapter 5).

setof: pattern:

$$setof(variable_1, quant\_conj\_list, variable_3) \text{ or}$$

$$setof(variable_1 : variable_2, quant\_conj\_list, variable_3) \text{ or}$$

$$setof(variable_1 : [variable_2], quant\_conj\_list, variable_3)$$

*variable*3 must be different from $variable_1$, $variable_2$ and may not appear within *quant_conj_list*. *setof* is used to create a set (*variable*3) whose elements satisfy *quant_conj_list*.

The following three predicates have special forms:

\+: $\backslash+(conjunction\_list)$ Non-satisfiability.

$>$: $variable > number$ The *variable* corresponds to an LQL number. *number* must not be a variable. *variable* must also appear as an argument of a predicate instance, conjoined to and preceding the $>$ instance, excluding *setof* arguments, first *length* arguments, second *min*, *max*, *tot*, *av* arguments, \+ arguments, *greater_than* and *less_than* arguments, and other $<$ or $>$ arguments.

$<$ and $>$ are similar to *greater_than* and *less_than*. Notice, however, the different predicate patterns.

$<$: Same argument pattern as $>$.

3.3.10 Logical queries

There are four forms of LQL logical queries:

yes/no queries: pattern:

$$answer([\,]) :- conjunction\_list$$

E.g: The question '*Is there some ocean that does not border any country?*' could be represented by the following LQL expression:

```
answer([]) :- ocean(_1),
              \+((country(_2),
                  borders(_1,_2))
```

simple enumeration queries: pattern:

$$answer([Var]) :- conjunction\_list$$

E.g: The question '*What is the largest country in Europe?*' could be represented by the LQL expression:

```
answer([_1]):- setof(_2:_3,
                     (country(_3),
                      has_area(_3,_2),
                      in(_3,europe)),
                     _4),
               max(_1,_4)
```

and the question '*What are the european countries?*' could be represented by the expression:

```
answer([_1]):- setof(_2,
                     (country(_2),
                      in(_2,europe)),
                     _1)
```

complex enumeration queries: pattern:

$$answer([Var_1, Var_2]) :- conjunction\_list$$

E.g: The question'*What is the average area of the countries in each continent?*' could be represented as:

```
answer([_1,_2]):- continent(_1),
                  setof(_3:[_4],
                        (area(_3,_4),
                         country(_4),
                         in(_4,_1)),
                        _5),
                  av(_2,_5)
```

cardinality queries: pattern:

$$answer([Var_1\#Var_2]) :- conjunction\_list$$

These queries are similar to complex enumeration queries. However, only Var_1 is reported to the user. The intention is to store internally Var_2, so that it can be used to answer possible follow-up questions. Cardinality queries are typically produced by '*How many ... ?*' questions. For example, the question '*How many countries does the Danube flow through?*' is represented as:

```
answer([_13138#_13139]):-
   setof(_12336,
         (flows(danube,_12336),
          country(_12336)),
         _13139),
   length(_13139,_13138)
```

Only the contents of `_13138` (the cardinality of the set) are reported to the user. The contents of `_13139` (the set of countries the Danube flows through) could be stored internally, to help the system answer follow-up questions, such as '*What are the names of the countries?*'. However, this feature is not implemented in Masque/PRO and Masque/SQL.

3.4 The semantics of LQL

3.4.1 Interpretation

When using LQL to express questions about a world, a suitable interpretation of LQL must be defined.

Consider for example the question *'Which countries border Italy?'*. The corresponding LQL query is:

```
answer([_10]):- country(_10),
                borders(_10, italy)
```

Although the meaning of the LQL query may seem obvious to a human, it is not formally defined, until `country(_10)` is explicitly defined to mean *'the world entity for which* `_10` *stands is a country'*, and `borders(_10, italy)` is explicitly defined to mean *'the world entity for which* `_10` *stands borders Italy'*.

The problem is that the meaning of LQL symbols (such as $italy$, $ba52$) and the meaning of domain dependent predicates (such as $country/1$, $borders/2$) may be different in each domain. Thus their meaning cannot be hard-wired into the LQL definition, as in the case of numbers and built-in predicates, the meaning of which is always the same and specified in the LQL definition.

To formally define the meaning of all possible LQL queries with respect to a particular world, every LQL symbol must be 'linked' to a world entity, and every domain-dependent predicate must be linked to a world relation. This 'linking', possibly different in each domain, is provided by a function, called *interpretation*, which is formally defined in the following paragraphs.

- Let E be the set of world entities.
- Let S be the set of world entity-sets.
- Let R be the set of world entity-relations.
- Let $EPSE$ be the set of world EPSE relations.

- Let A be the set of LQL atoms.
- Let V be the set of LQL variables.
- Let SY be the set of LQL symbols.
- Let PR be the set of LQL pairs (not containing variables).
- Let LS be the set of LQL sets (not containing variables).
- Let DP_1 be the set of domain-dependent predicates introduced by nouns with no complements and qualitative adjectives.
- Let DP_2 be the set of domain-dependent predicates introduced by verbs, prepositions, quantitative adjectives or nouns having entity complements.
- Let DP_B be the set of type-B domain dependent predicates.

An interpretation I is a function:

$$I : SY \cup DP_1 \cup DP_2 \cup DP_B \rightarrow E \cup S \cup R \cup EPSE$$

such that:

- $\forall s \in SY \;\; I(s) \in E$
- $\forall p \in DP_1 \;\; I(p) \in S$
- $\forall p \in DP_2 \;\; I(p) \in R$, such that the arities of p and $I(p)$ are the same
- $\forall p \in DP_B \;\; I(p) \in EPSE$
- I is $1-1$ (i.e. $\forall p_1, p_2 \in SY \cup DP_1 \cup DP_2 \cup DP_B \;\; p_1 \neq p_2 \rightarrow I(p_1) \neq I(p_2)$)

Note that the interpretation function is not defined over LQL elements such as numbers or built-in predicates, because their meaning is domain-independent.

3.4.2 Variable assignment

Variables may appear in the place of any LQL atom, pair or set. The meaning of an LQL variable is defined by a variable assignment.

A variable assignment is a function $g : V \rightarrow A \cup PR \cup LS \cup \{nil\}$.

A variable var for which $g(var) = nil$, where g is a variable assignment, is called an *uninstantiated variable* with respect to g.

Let $LQLE$ be the set of all LQL expressions and all fragments of LQL expressions. For every variable assignment g we define a function $g^* : LQLE \rightarrow LQLE$, such that $\forall\, e \,\epsilon\, LQLE$, $g^*(e)$ is the expression created from e, by replacing every variable occurrence var of e by $g(var)$.

A variable assignment g_2 is *compatible* with a variable assignment g, iff for every variable var that is instantiated wrt g, $g_2(var) = g(var)$ or $g_2(var) = nil$.

A variable assignment g is *non-redundant* with respect to a conjunction list L, iff all variables

- appearing only within \+ instances in L or
- appearing only within 1st argument positions of $setof$ in L or
- appearing only within 2nd argument positions of $setof$ in L or
- not appearing within L

are uninstantiated wrt g. Intuitively, variables that only appear within first or second argument positions of a $setof$ instance are only used to 'compute' the third argument; they are auxiliary variables, local to the $setof$ instance, and if they get a binding during the $setof$ evaluation, this binding should not affect other predicate instances, with which the $setof$ forms a conjunction list. Similar restrictions apply for variables that only appear within a \+ instance.

A variable assignment g is *sufficient* with respect to a conjunction list L, iff all variables of L, excluding variables

- appearing only within \+ instances in L
- appearing only within 1st argument positions of $setof$ in L
- appearing only within 2nd argument positions of $setof$ in L

are instantiated wrt g.

A variable assignment g is a *proper* variable assignment for a conjunction list L, iff it is non-redundant and sufficient with respect to L.

3.4.3 Evaluation

LQL expressions can be evaluated, returning a result which is the correct answer to the corresponding question.

The evaluation of an LQL expression e with respect to an interpretation I, written $Eval_I(e)$, is defined as follows. (In the following lines '*wrt g*' means '*with respect to a variable assignment g*'.)

- Numbers evaluate to themselves.
- nil evaluates to itself.
- For every symbol s, s evaluates to $I(s)$.
- For every LQL pair $first : second$ or $first : [second]$, the pair evaluates wrt g to the entity-pair $(Eval_I(g^*(first)), Eval_I(g^*(second)))$.
- For every LQL set $[element_1, \ldots, element_n]$, the set evaluates wrt g to $\{Eval_I(g^*(element_1)), \ldots, Eval_I(g^*(element_n))\}$. The LQL set $[\,]$ evaluates to $\{\}$.
- A predicate instance $f(arg_1, \ldots, arg_n)$ of a domain-dependent predicate p, evaluates to $true$ wrt g iff
  - all variables appearing in $f(arg_1, \ldots, arg_n)$ are instantiated wrt g and
  - $(Eval_I(g^*(arg_1)), \ldots, Eval_I(g^*(arg_n))) \in I(p)$.

  The predicate instance evaluates to $false$ otherwise.
- '$conj\_list_1, conj\_list_2$' evaluates to $true$ wrt g, iff $conj\_list_1$ and $conj\_list_2$ both evaluate to $true$ wrt g; it evaluates to $false$ otherwise.
- A quantified conjunction list evaluates to $true$ wrt g, iff its body evaluates to $true$ wrt g; it evaluates to $false$ otherwise.
- $true$ evaluates to itself.
- $length(set\_var, len)$ evaluates to $true$ wrt g iff $Eval_I(g^*(len))$ is the cardinality of $Eval_I(g^*(set\_var))$; it evaluates to $false$ otherwise.
- $is(var_1, var_2)$ evaluates to $true$ wrt g iff $g(var_1) = g(var_2) \neq nil$. It evaluates to $false$ otherwise.

- $min(arg_1, arg_2)$ evaluates to $true$ wrt g iff

  - $g^*(arg_2) \neq [\,]$ and
  - $g^*(arg_1) = atom \, \epsilon \, A$ and
  - $g^*(arg_2)$ has the form $[number_1\!:\!atom_1, \ldots, number_n\!:\!atom_n]$
  - $number_1, \ldots, number_n$ are LQL numbers and
  - $atom_1, \ldots, atom_n \, \epsilon \, A$ and
  - $number = min\{number_1, \ldots, number_n\}$ and
  - $number\!:\!atom$ appears within $g^*(arg_2)$

  It evaluates to $false$ otherwise.

- The evaluation of max is similar.

- $av(arg_1, arg_2)$ evaluates to true wrt g iff

  - $g^*(arg_2) \neq [\,]$ and
  - $g^*(arg_1) = number$, where $number$ is an LQL number and
  - $g^*(arg_2)$ has the form $[number_1\!:\!atom_1, \ldots, number_n\!:\!atom_n]$
  - $number_1, \ldots, number_n$ are LQL numbers and
  - $atom_1, \ldots, atom_n \, \epsilon \, A$ and
  - $number$ is the average of $number_1, \ldots, number_n$.

  It evaluates to $false$ otherwise.

- The evaluation of tot (total, sum) is defined similarly.

- $variable > number$ evaluates to $true$ wrt g, iff

  - $g(variable)$ is an LQL number and
  - number is an LQL number (not a variable) and
  - $g(variable) > number$

  It evaluates to $false$ otherwise.

- The evaluation of $<$ is defined similarly.

- $greater\_than(variable_1, variable_2)$ evaluates to $true$ wrt g, iff:

  - $g(variable_1) = number_1$ and
  - $g(variable_2) = number_2$ and
  - $number_1, number_2$ are LQL numbers and
  - $number_1 > number_2$

  It evaluates to $false$ otherwise[2].

- The evaluation of $less\_than$ is defined similarly.

- $\backslash+(conjunction\_list)$ evaluates to $true$ wrt g, iff there is no variable assignment g_2 compatible with g, such that $conjunction\_list$ evaluates to $true$ wrt g_2; it evaluates to $false$ otherwise.

- $setof(arg_1, quant\_conj\_list, set)$ evaluates to $true$ wrt g iff $Eval_I(g^*(set))$ is the set of all $Eval_I(g_2^*(arg_1))$ (for all possible g_2), such that:

  - g_2 is a variable assignment and
  - the body of $quant\_conj$ evaluates to $true$ wrt g_2 and
  - g_2 is compatible with g and
  - $Eval_I(g^*(set))$ is not the empty set

  It evaluates to $false$ otherwise.

- $answer([]) : -conjunction\_list$ evaluates to $true$, iff for some variable assignment g,

---

[2]In Masque/PRO, where predicate instances are treated as Prolog goals, a failed $greater\_than$ or $less\_than$ goal has the side-effect of creating an internal 'ignorance note', that helps the system report a helpful message to the user. Also, in Masque/PRO $<$, $>$, $greater\_than$, and $less\_than$ can also be used to compare arbitrary Prolog terms, since the comparisons are carried out using the Prolog operators @> and @<.

- $conjunction\_list$ evaluates to $true$ wrt g and
- g is a proper variable assignment for $conjunction\_list$.

It evaluates to $false$ otherwise.

- $answer([variable]) :- conjunction\_list$ evaluates to the set:

$$\{Eval_I(g(variable)) \mid g \text{ is a variable assignment} \wedge$$
$$conjunction\_list \text{ evaluates to } true \text{ wrt } g \wedge$$
$$g \text{ is proper for } conjunction\_list \wedge$$
$$variable \text{ is instantiated wrt } g\}$$

iff the above set is not empty; it evaluates to $false$ otherwise.

- $answer([variable_1, variable_2]) :- conjunction\_list$ evaluates to the set:

$$\{(Eval_I(g(variable_1)), Eval_I(g(variable_2))) \mid g \text{ is a variable assignment} \wedge$$
$$conjunction\_list \text{ evaluates to } true \text{ wrt } g \wedge$$
$$g \text{ is proper for } conjunction\_list \wedge$$
$$variable_1, variable_2 \text{ are instantiated wrt } g\}$$

iff the above set is not empty; it evaluates to $false$ otherwise.

- $answer([variable_1 \# variable_2]) :- conjunction\_list$ evaluates to the set:

$$\{(Eval_I(g(variable_1)), Eval_I(g(variable_2))) \mid g \text{ is a variable assignment} \wedge$$
$$conjunction\_list \text{ evaluates to } true \text{ wrt } g \wedge$$
$$g \text{ is proper for } conjunction\_list \wedge$$
$$g(variable_1) \text{ is an unsigned integer} \wedge$$
$$variable_2 \text{ is instantiated wrt } g\}$$

See also comments about *cardinality queries* in section 3.3.10.

3.5 Shortcomings of the LQL specification presented

- Masque/PRO contains a logical query optimiser [Auxerre & Inder 86]. The optimiser transforms the logical queries by:

  - reordering the predicate instances within conjunction lists
  - inserting Prolog cuts (realised as `deterministic_call` instances).

  to allow the Prolog interpreter execute the logical queries more efficiently [Warren 81]. This chapter has described the logical expressions Masque generates when the logical query optimiser is disconnected. The optimiser is not used in Masque/SQL, the version of Masque produced during this project.

- The logical queries generated by Masque/PRO may also contain instances of a predicate `must_satisfy`(*predicate_instance*). Masque/PRO evaluates `must_satisfy` instances by simply calling *predicate_instance* as a Prolog goal. If, however, the call fails, an internal '*ignorance note*' is kept, that helps the system produce a reasonably helpful message to the user [Sentance 89]. Masque/SQL does not generate `must_satisfy` instances.

- Masque/PRO allows upper-case letters in predicate functors; on the contrary, the LQL definition of this document, which is the definition used in Masque/SQL, doesn't. The decision not to allow upper-case letters in predicate functors was taken, in order to make the translation from LQL to SQL (see later chapters) easier: SQL table names are not case-sensitive. The case restriction shouldn't cause any serious problems, since no built-in predicate contains upper-case letters in its functor, and the knowledge-engineer can easily choose domain-dependent predicate functors with no upper-case letters.

- Masque provides a '*proper names preprocessor*' [Lindop 86]. The preprocessor allows proper names in the English question to be transformed into internal atoms. This feature is particularly useful in cases where a proper name cannot be written as an LQL atom. For example, the user might ask: '*where is the United Kingdom?*'. We would like Masque to generate an LQL query of the form:

```
answer([_12]):- place(_12),
                in(united_kingdom, _12).
```

However, Masque would not be able to infer that '*United Kingdom*' corresponds to the internal atom *united_kingdom*. Masque's proper names preprocessor provides a mechanism to link names such as '*United Kingdom*' or '*United States Of America*' to internal atoms such as '*united_kingdom*' and '*usa*'.

In fact, the proper names preprocessor allows proper names to be transformed into arbitrary Prolog terms. For example, in the '*fly85*' domain, a sample domain for Masque/PRO, the question '*what airports does sm722 link?*' is preprocessed into '*what airports does* $f(sm722, [aberdeen, dundee])$ *link?*', and the generated LQL query is:

```
answer([_15]):- airport(_15),
                links(f(sm722, [aberdeen,dundee]), _15)
```

The above example shows that the proper names preprocessor may cause arbitrary Prolog terms to appear in predicate argument positions. On the contrary, the LQL specification of this chapter allows a very limited number of term types to appear in predicate argument positions (e.g: LQL atoms, LQL sets etc). This is done in order to keep the LQL specification simple, and also simplify the process of translating from LQL into SQL (see later chapters), where complex terms are not allowed as table attribute values.

The example also shows a case where the proper names preprocessor issues a retrieval request towards the world database (see following chapter). This extra database access is problematic for the Masque/SQL evaluation method that will be described in the following chapters.

Although, the proper names preprocessor is still available in Masque/SQL, the preprocessor should be used carefully, making sure that proper names are transformed into terms allowed by the LQL specification of this chapter.

3.6 Summary

Masque translates English questions into expressions of a logical language called LQL. An LQL expression specifies exactly what Masque understands to be the meaning of the corresponding English question.

Although LQL is a subset of Prolog, LQL expressions are not arbitrary Prolog code. Only certain kinds of predicates and terms may appear within an LQL expression, and there are only few possible forms an LQL query may have. The semantics of LQL are also 'weaker' than the semantics of Prolog. For example, the ordering of the predicate instances within a conjunction list is (with few exceptions) not significant, and an LQL query need not necessarily be evaluated using depth-first search.

This chapter has provided a detailed description of the LQL syntax and semantics. Such a description is a necessary step, in order to design a method for the evaluation of LQL queries against relational databases (see following chapters).

Chapter 4

Evaluating Logical Queries Against a Relational Database

4.1 Introduction

The previous chapter described the logical queries generated by Masque. Masque transforms each English question into an LQL query. The LQL query expresses precisely what Masque understands to be the meaning of the corresponding English question, and hence specifies what Masque considers to be a valid answer to the English question. However, the logical query itself is not an answer to the user's question, nor does it specify the strategy that should be used to generate the answer. The user's question is answered by *evaluating* the corresponding logical query.

To evaluate a logical query, one needs to be able to access the world to which the query refers. Consider for example the question '*is any cube on top of a cube?*', referring to a world of small cubes and pyramids. The corresponding LQL query is:

```
answer([]):- cube(_10),
             cube(_20),
             on(_20,_10)
```

To evaluate this query, a computer system would need to be able to identify the cubes in the relevant world, and decide if any cube is on top of another. Such an approach would require advanced sensing and reasoning abilities. Instead, a simpler approach is adopted in Masque: The system consults a database, called *the world database*, which holds all the necessary facts about the relevant world, and the logical query is *evaluated against the database.* The world database could be updated by people or other computer systems.

In the previous example, a system using a relational world database could consult two database tables of the form:

OBJECT_REL	
OBJECT	SHAPE
1	cube
2	pyramid
3	cube

ON_REL	
UNDER	ON_TOP
1	2
3	1

The left table informs the system that objects 1 and 3 are both cubes, while object 2 is a pyramid. The right table states that object 2 is on top of object 1, and that object 1 is on top of object 3.

Of course the search strategy must also be specified: should the system first identify all cubes, and then check if any of them is on top of another? Should it first find all object pairs, where one pair is on top of another, and then check if both objects are cubes?

This chapter discusses several methods that can be used to evaluate logical queries (not necessarily LQL queries) against a *relational* database. Although a logical query could probably be evaluated more naturally against a deductive database [Minker 88], it is assumed that the world database is relational, since most modern commercial databases adopt the relational model.

The purpose of the chapter is to present some of the ideas that were used to design the logical query evaluation method of Masque/SQL, and to suggest alternative strategies that might have been used.

- Section 2 describes the evaluation method of Masque/PRO. In Masque/PRO the Prolog database is also used as the world database. Although the Prolog database is not relational (and not even a database with the current computer science meaning), the Masque/PRO evaluation method serves as a useful introduction to the other evaluation techniques presented in this chapter.

- Section 3, presents an extension of the Masque/PRO evaluation method: providing a transparent mechanism, that allows Prolog programs to access facts stored in a commercial relational database, as if they were stored in the Prolog database. I will call the resulting evaluation method '*the extended Masque/PRO evaluation method*'.

- Although LQL is a subset of Prolog, there is no need to evaluate LQL queries by executing them as Prolog programs. Section 4 discusses alternative methods, that translate the logical queries into SQL code. These methods are more efficient than the extended Masque/PRO evaluation method. Two systems that use this approach, a natural language front-end developed at Essex University, and IBM's LanguageAccess are presented.

- Translating LQL queries into SQL provides an efficient and portable way to interface Masque to almost any relational database. However, this method restricts the knowledge available during the question answering to the information that can be retrieved from the relational database. In that sense, the extended Masque/PRO evaluation method, although less efficient, is more powerful. Section 5 presents an alternative evaluation approach, based on a more powerful coupling between Prolog and the relational database, that seems to combine the advantages of all previous methods.

- Finally, section 6 summarises the chapter.

4.2 Evaluating LQL queries against the Prolog database

Masque/PRO treats LQL queries as Prolog programs. (Recall that LQL is actually a subset of Prolog.) An LQL query matching the Prolog pattern:

```
answer(Answer_list) :- Conjunction_list
```

is evaluated by executing the Prolog program:

```
setof(Answer_list, (Conjunction_list), Answer).
```

and by reporting the contents of `Answer` to the user. The actual code of Masque/PRO is more complicated, but the general idea is the same.

The extra `setof` call is introduced to ensure that we get *all* the possible answers, and that the user is only informed about bindings of variables appearing in the `Answer_list`.

For example, the question '*where is Paris?*' produces the LQL query:

```
answer([_10449]):- in(paris,_10449),
                   place(_10449)
```

which is evaluated by executing the Prolog call:

```
setof([_10449], (in(paris,_10449),
                 place(_10449)),
      Answer).
```

The result produced by the Prolog interpreter is:

```
Answer = [[france], [western_europe], [europe]]
```

(Masque/PRO reports the answer in a prettier format.)

Similarly, the question '*what is the total area of the countries in Europe?*' produces the LQL query:

```
answer([_11912]):-
        setof(_11915:[_11916],
              (area(_11915,_11916),
               country(_11916),
               in(_11916,europe)),
              _11914),
        tot(_11912,_11914)
```

which is evaluated by executing the Prolog call:

```
setof([_11912],
      (setof(_11915:[_11916],
             (area(_11915,_11916),
              country(_11916),
              in(_11916,europe)),
       _11914),
       tot(_11912,_11914)),
      Answer).
```

and reporting the result: `Answer = [[1866]]`

All LQL built-in predicates that are not also Prolog built-in predicates (such as `tot` in the above example) are implemented internally in the Masque/PRO code. Domain-dependent predicates need to be implemented as normal Prolog predicates in the '*interface with data file*' whenever a new domain is defined. (See [Auxerre & Inder 86] for more details.)

The LQL query of the first example contained instances of the domain dependent predicates `in` and `place`. In the 'interface with data' we could have:

```
in(paris,france).
in(france,western_europe).
in(western_europe,europe).
in(X,Y):- in(X,Z),
          in(Z,Y).
place(X):- in(X,_); in(_,X).
```

Thus, knowledge about the world is stored in the Prolog database. The Prolog implementations of domain-dependent predicates may be as complex as necessary, and they may contain calls to other complex Prolog programs.

This approach has three main advantages:

- It allows the world-database to be any Prolog system. The actual form of the world database is not important, as long as it can 'respond' to all possible types of domain-dependent predicate calls (e.g. calls with unbound variables, backtracking etc).

- The interaction of Masque with the world-database is simple, since they are both implemented in Prolog.

- It provides a ready-made evaluator for LQL queries, namely the Prolog interpreter.

However:

- Since LQL queries are executed as Prolog programs, we rely on Prolog's depth-first, backtracking, left-to-right strategy to evaluate LQL queries. This strategy may not be the most appropriate in some domains.

  Note, however, that Masque/PRO provides a logical query optimiser, that reorders predicate instances within LQL conjunction lists, and inserts Prolog cuts, to allow the Prolog interpreter to execute the query more efficiently. See [Warren 81], [Auxerre 86], [Auxerre & Inder 86].

- Since LQL queries are evaluated only against the Prolog database, this approach does not allow the use of commercial external databases. One might need to use Masque as a front-end to a commercial external database, in order to access existing information, or to ensure that the world-database has properties such as security, concurrency, durability etc. For example, we might want to use many copies of Masque, to allow simultaneous queries against a sales database, that is continuously updated.

4.3 Treating the relational database as an extension of the Prolog database

Rows stored in a relational database table can be seen as a collection of Prolog facts. For example, the database table:

LOCATIONS	
PLACE	LOCATION
paris	france
paris	europe
paris	western_europe
athens	greece
athens	europe

is logically equivalent to the Prolog facts

```
locations(paris, france).
locations(paris, europe).
locations(paris, western_europe).
locations(athens, greece).
locations(athens, europe).
```

It should, therefore, be possible to provide a 'link' between Prolog and a relational database, so that the Prolog interpreter can use the facts stored in the relational database, as if they were stored in the Prolog database. Such a link would allow Prolog to be used as a database query language.

Prolog has the power of a Turing Machine. On the contrary, SQL and other similar relational database query languages are less powerful. This will be demonstrated by a transitive closure example later. So, using Prolog as a database query language, would allow more queries to be answered. See [Ceri *et al* 90] for a discussion of logical languages for databases.

A number of ways to link Prolog to relational databases have been proposed. In [Lucas 88], the writer describes a transparent interface that allows rows stored in tables of a relational database to be treated exactly as if they were ground facts in the Prolog database. The Prolog programmer needs only to provide declarations of the form:

$$\texttt{external}(predicate/arity, table\_name).$$

that map Prolog predicates to database tables.

Designing such an interface is far from trivial. The 'external' predicates must behave exactly as ordinary Prolog predicates. They must backtrack when needed, to provide alternative solutions, and they must handle calls with both bound and free variables.

Interfaces based on Lucas' description are commercially available for a variety of commercial relational databases and Prolog implementations. It should be noted, however, that the following discussion is based on the interface description of [Lucas 88], and that no actual implementation of the interface has been tested during this project.

Such an interface, would make interfacing Masque to a relational database almost trivial:

- The evaluation method of Masque/PRO could still be used, i.e. LQL queries would be executed as Prolog programs.

- The knowledge engineer would be able to implement some of the domain-dependent predicates in ordinary Prolog, and some others by linking them to tables/views of the relational database. For example, given the table `locations`, described above, the domain-dependent predicates `place/1` and `in/2` could have been implemented in the '*interface with data*' as:

  ```
  external(in/2, locations).
  place(X):- in(X,_); in(_,X).
  ```

  Note that, since the '*interface with data*' can still be consulted by Prolog as before, the Masque/PRO code could be used with no modifications.

This approach could have some significant advantages:

- It allows Masque to use knowledge stored in relational databases.

- No modification of the Masque/PRO code is needed, provided that the interface code is available.

- Some of the domain-dependent predicates can still be (partially or completely) implemented in normal Prolog. Thus the full power of Prolog is available when defining the '*interface with data*'. The knowledge engineer can use Prolog code to complement the data stored in the relational database, providing additional knowledge, that is not stored or cannot be stored in the relational database.

  Consider for example the domain-dependent predicate

  ```
  in(Contained,Contains)
  ```

  of a geography domain. Clearly, we want `in/2` to provide the effect of a transitive closure. If the facts:

```
in(area1, athens).
in(athens, greece).
in(greece, europe).
```

are true, then we expect the following facts also to be true:

```
in(area1, greece).
in(area1, europe).
in(athens, europe).
```

The brute-force solution of defining `in/2` as `external(in/2, IN_TABLE)`, and having in the relational database a table:

IN_TABLE	
CONTAINED	CONTAINS
area1	athens
athens	greece
greece	europe
area1	greece
area1	europe
athens	europe

is problematic:

- The last three rows of `IN_TABLE` are redundant. They can be logically inferred from the first three.
- The number of redundant rows can increase dramatically in more complex domains. Consider, for example, the case of a chemical plant. We want to be able to determine which parts of the plant are subparts of other parts. Keeping a table:

PART_OF	
CONTAINED	CONTAINS
plant	building1
plant	building2
...	...
building1	system1
building1	system2
...	...
system1	engine1
...	...
engine1	pipe1
plant	pipe1
building1	pipe1

is obviously non-efficient. Ensuring the data integrity of the database is also very difficult. If *pipe*1 is replaced in *engine*1 by *pipe*10, then all rows:

plant	pipe1
building1	pipe1
system1	pipe1
engine1	pipe1

need to be updated.

Clearly, what we need is a rule of the form:

$$subpart(X,Y) \wedge subpart(Y,Z) \rightarrow subpart(X,Z)$$

In Masque/PRO with Lucas' interface, we could simply have a database table:

$$\texttt{IMMEDIATE\_PART\_OF}(contains,\ contained)$$

and define `part_of/2` as:

```
external(immediate_part_of/2, IMMEDIATE_PART_OF).
part_of(X,Y):- immediate_part_of(X,Y).
part_of(X,Z):- immediate_part_of(X,Y),
               part_of(Y,Z).
```

The approach of using Masque/PRO complemented by an interface of the kind described above interface could have some very desirable properties. There is, however, a potentially significant disadvantage: a system evaluating LQL queries using this method could be extremely inefficient. Consider the LQL query:

```
answer([_1]):- country(_1),
               european(_1),
               borders(_1,greece)
```

where the relational database contains the tables:

$$\texttt{COUNTRY\_REL}(country\_code,\ country\_name,\ continent)$$

$$\texttt{BORDERS\_REL}(country\_code_1,\ country\_code_2)$$

and where the '*interface with data*' contains:

```
external(country_pred/3, COUNTRY_REL).
external(borders_pred/2, BORDERS_REL).
country(C):- country_pred(_, C, _).
european(C):- country_pred(_, C, europe).
borders(C1,C2):- country_pred(Code1, C1, _),
                 country_pred(Code2, C2, _),
                 borders_pred(Code1, Code2).
```

A system executing the LQL query as a Prolog program would:

- first issue a request towards the commercial database to find a possible country-value for `_1`
- then issue a request towards the commercial database to check if the country found is also european
- then issue more requests towards the commercial database to ensure that the country borders Greece.
- If, say, after the third step the country-value proved to be inappropriate, the system would have to backtrack to the first step, to select another country. It would issue a new request towards the database to get a new country-value, then issue requests to ensure that the new country is european and that it borders Greece, and so on.

Given that requests towards the commercial database are usually very expensive, in terms of execution time, one can easily understand that the overall performance could be very poor.

The inefficiency becomes more clear if one realises that in the previous example we could get all the results by using the single SQL query:

```
SELECT B.COUNTRY_NAME
FROM BORDER_REL, COUNTRY_REL A, COUNTRY_REL B
WHERE A.COUNTRY_NAME = 'greece'
  AND A.COUNTRY_CODE = COUNTRY_CODE1
  AND B.COUNTRY_CODE = COUNTRY_CODE2
```

A well-designed Prolog to relational database interface could reduce the number of requests towards the relational database, by using SQL cursors [ING90a] and a clever strategy to load rows from the relational to the Prolog database [Ceri *et al* 89]. However, the real problem of this approach would still remain unsolved:

The relational management system (RDBMS) of the world database is used just to get partial results. The Prolog interpreter is then used to join the partial results, using its limited top-down backtracking strategy. Instead, an overall request towards the RDBMS should be issued, and the RDBMS should be left to find all possible solutions, using its own specialised optimisation and planning techniques. For example, a commercial RDBMS could use relational algebra transformations, indices, statistics and clustering information to reduce retrieval time.

Of course Lucas' interface was not designed having LQL queries in mind. It provides a general mechanism to make information stored in relational databases available to Prolog programs. In an arbitrary Prolog program, it may not be possible to formulate an overall query towards the RDBMS, because the 'external' predicates may be interleaved with ordinary Prolog predicates.

In the case of Masque, however, the problem seems to be that, although we want to be able to express logical queries using a Prolog-like language, we do not want to constrain the query answering process to Prolog's strategy.

Such considerations have led to the development of Datalog, a Prolog-like language, based on strict Horn clause logic [Ceri *et al* 90]. Datalog is insensitive to the order in which rules appear in a program, and to the order of the predicates in the body of each rule. However, no Datalog support is available for most commercial relational databases, and hence Datalog will not be discussed further.

4.4 Translating logical queries into SQL

The logical query evaluation methods presented so far answer the user's question by executing the logical query as a Prolog program. Instead, the methods discussed in this section transform the logical query into SQL code, and then let the RDBMS execute the SQL code to answer the user's question. This way, a single overall query is transmitted to the relational database, which can be processed more efficiently by the RDBMS.

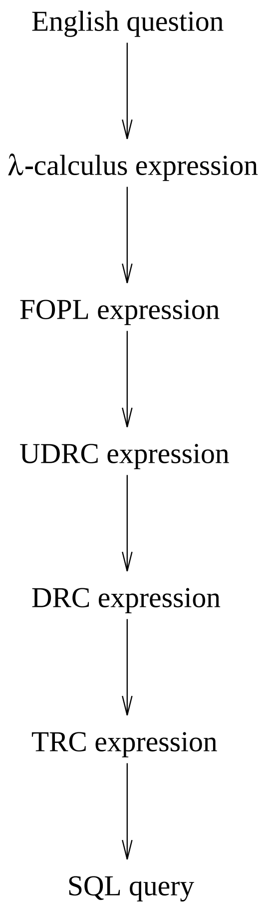

Figure 4–1: Transformations used in the Essex front-end

4.4.1 A principled approach: The Essex system

In [De Roeck *et al* 90], [Lowden *et al* 91], [De Roeck *et al* 91], a natural language front-end for relational databases, developed at the University of Essex, is described. The Essex system answers English questions by subsequently transforming them to λ-calculus, first order predicate logic (FOPL), relational calculus and finally SQL expressions. Figure 4–1 shows the transformations used.

Suppose the user asks the question '*What is Fred's age?*'. The example is borrowed from [Lowden *et al* 91], and assumes the database table:

$$EMPL(name, age, sal, dept)$$

Using a context-free grammar, where each production rule has an associated compositional semantics rule (see [Pereira & Shieber 87] for more information about compositional semantics and λ-calculus), the question is translated into the λ-calculus expression:

$$\Xi x.of\ x\ AGE\ FRED \wedge be\ fv\ x$$

The λ-calculus expression is then transformed into a FOPL expression, applying Turner's Property Theory axioms [De Roeck *et al* 91]:

$$\exists\ x\ of(x, AGE, FRED) \wedge be(fv, x)$$

Note that fv is a free variable; the Essex system attempts to report all possible bindings for each free variable.

Each predicate appearing in the generated FOPL expressions must have been linked to a Universal Domain Relational Calculus (UDRC) expression. This is similar to the domain-dependent predicate implementations the knowledge engineer has to provide in the Masque '*interface with data*'.

UDRC is similar to Domain Relational Calculus (DRC) [Ullman 88], except that all variables are assumed to range over a single universal domain.

Assuming that the predicates of and be have been linked to UDRC expressions as follows:

$$\begin{aligned} of(x, AGE, y) &\longleftrightarrow\ < y, x, *, * >\ \epsilon\ EMPL \\ be(x, y) &\longleftrightarrow\ x = y \end{aligned}$$

the FOPL expression is translated into the UDRC expression:

$$\{fv \mid \exists\ x\ [< FRED, x, *, * >\ \epsilon\ EMPL\ \wedge\ fv = x]\}$$

The asterisks denote existentially quantified anonymous variables. Note that the attributes of each relation are assumed to be ordered. Therefore, $< y, x, *, * >$ means $EMPL.name = y$ and $EMPL.age = x$.

The UDRC expression is then turned into a Domain Relational Calculus (DRC) expression [Ullman 88]. In DRC each variable ranges over a specific table attribute. We can easily transform UDRC expressions into DRC, by observing the table attributes to which the UDRC variables refer. For example, in $< FRED, x, *, * >$,

x corresponds to the second attribute of the $EMPL$ table, i.e. it should range over $EMPL[age]$.

The UDRC expression of our example is translated into the DRC expression:

$$\{fv \,\epsilon\, EMPL[age] \mid \exists\, x \epsilon\, EMPL[age]\, [< FRED, x, *, * > \epsilon\, EMPL \;\wedge\; fv = x\}$$

The DRC expression is then translated into a Tuple Relational Calculus (TRC) expression [Ullman 88]. In TRC variables correspond to table rows. [Ullman 88] and [Lowden *et al* 91] describe algorithms to translate from DRC to TRC.

The DRC expression of our example is transformed into the TRC expression:

$$\{fv.age \mid fv \,\epsilon\, EMPL \mid \exists\, x \,\epsilon\, EMPL\, [\exists\, tuple \,\epsilon\, EMPL$$

$$[tuple.name = FRED \;\wedge\; tuple.age = x.age] \;\wedge\; fv.age = x.age]\}$$

Since SQL is a version of TRC, the mapping from TRC to SQL is relatively simple [Lowden *et al* 91]. The TRC expression of our example is mapped to the SQL query:

```
SELECT DISTINCT fv.age
FROM EMPL fv
WHERE EXISTS (SELECT *
              FROM EMPL x
              WHERE EXISTS (SELECT *
                            FROM EMPL tuple
                            WHERE tuple.name = 'FRED' AND
                                  tuple.age = x.age )
                AND fv.age = x.age )
```

Clearly, the Essex system uses a principled evaluation strategy. The English question is transformed into SQL using consecutive transformations. At each stage, a formal and precisely defined logical language is used. It is thus possible to define concisely the evaluation process, and prove its correctness.

There is, however, a limitation in the Essex system: Each question is answered by executing a single SQL query. This means that only knowledge that can be retrieved by the SQL query is available during the question answering. Therefore, returning to the chemical plant example of section 4.3, the Essex system would not be able to answer a question like '*What are all the subparts of engine5*', unless the transitive closure were to be stored explicitly in the database. The question refers to the transitive closure of a table, and there is no way to compute a table's transitive closure with a single SQL query [Ullman 88][1]. Note that the system will probably be able to generate the correct FOPL expression:

$$subpart(engine5, fv)$$

The problem is that the transitive closure query, corresponding to the predicate *subpart*, cannot be expressed in relational calculus.

At the heart of the problem is the the fact that the evaluation method restricts the query answering process to the knowledge retrieved by the SQL query. This is a general problem with all approaches that translate the logical queries to SQL code.

4.4.2 A more ad hoc approach: IBM's LanguageAccess

LanguageAccess is an IBM-licenced natural language interface for relational databases, available for English and German. [Ott 92] describes the logical query evaluation method used in LanguageAccess.

LanguageAccess uses two logical query languages, each corresponding to a different transformation stage. Here, only the latest stage language, called DBLF, is discussed.

DBLF is similar to LQL. Consider, for example, the question '*Which country belongs to neither the EC nor to an organisation that has not more than 10 members?*'. The example is borrowed from [Ott 92] and assumes the database relations:

$$\texttt{country\_info}(country, capital, population)$$
$$\texttt{organ\_info}(country, organisation)$$

---

[1]However, some non-standard SQL extensions can compute transitive closures in limited cases.

LanguageAccess would produce a DBLF query of the form (the actual DBLF query is slightly more complex):

```
query(report,
      set(y1,
          relation(country_info(country = y1)) &
         ~relation(organ_info(country = y1, organisation = 'EC')) &
         ~exist(y2,
               relation(organ_info(country = y1,
                                   organisation = y2)) &
               ~(10<count(set(y3,
                              relation(organ_info(country = y3,
                                                  organisation = y2))
                             ))))))
```

An equivalent LQL query could look like (Prolog variable names are used instead of LQL variables, to make the query easier to understand):

```
answer([Country_set]):-
   setof(Y1,
         (country_info(Y1, _, _, _, _),
         \+ (organ_info(Y1, 'EC')),
         \+ (organ_info(Y1, Y2),
             \+ (setof(Y3,
                       organ_info(Y3, Y2),
                       Members_set),
                 length(Members_set, No_of_members),
                 No_of_members > 10))),
         Country_set)
```

As in the case of LQL, DBLF uses the higher order predicate *set*, and variable unification is used to express joins between database tables.

LanguageAccess evaluates the DBLF queries by translating them into SQL. Actually, LanguageAccess uses a proprietary extended version of SQL, called SQL+. In [Ott 92] it is claimed that, using SQL+, some queries can be expressed more naturally than in ordinary SQL. An SQL+ query may contain many chained ordinary SQL queries. However, SQL+ doesn't seem to be more expressive than SQL. So, the limitations discussed in the case of the Essex system are probably valid for LanguageAccess, too.

Since DBLF is similar to LQL, it is interesting to study some of the techniques used in LanguageAccess to translate the DBLF queries into SQL. [Ott 92] discusses three techniques:

the join strategy: This is a technique to translate conjunction lists into SQL. Consider, for example, the conjunction list:

```
relation(country_info(country = y1)),
relation(organ_info(country = y1, organisation = 'EC'))
```

The conjunction list is scanned from left to right. For every conjunct encountered, the corresponding table is inserted into the `FROM` clause of the SQL query. The arguments of the predicates encountered generate conditions that are added into the `WHERE` clause of the SQL query.

In the conjunction list above,

```
relation(country_info(country = y1))
```

is encountered first. The SQL query becomes:

```
SELECT ...
FROM country_info
WHERE country = y1
```

Then:

```
relation(organ_info(country = y2, organisation = 'EC'))
```

is encountered, and the SQL query becomes:

```
SELECT ...
FROM country_info, organ_info
WHERE country_info.country = y1 AND
      organ_info.country = country_info.country AND
      organ_info.organisation = 'EC'
```

the negation strategy: This is a technique to translate negated logical expressions into SQL. Consider the negated logical expression:

```
~exist(y2, relation(organ_info(country = y2)))
```

First, the non-negated expression is translated into:

```
SELECT DISTINCT country
FROM organ_info
WHERE country = y2
```

Then, the negated logical expression is translated into:

```
SELECT ...
FROM ...
WHERE NOT EXISTS (SELECT DISTINCT country
                  FROM organ_info
                  WHERE country = y2 )
```

The actual negation strategy described in [Ott 92] is more complex.

the temporary relation strategy: In some cases, translating a logical query into a single SQL query may be quite complicated. Consider, for example, the question '*Is there a country which exports all products?*'. (Again, the example is taken from [Ott 92].)

The problem is that SQL doesn't provide an operator for universal quantification. So, we are forced to use two nested `NOT EXISTS`s. A possible SQL query would be:

```
SELECT x1.country
FROM export_info x1
WHERE NOT EXISTS
   (SELECT x2.product
    FROM export_info x2
    WHERE NOT EXISTS
       (SELECT x3.country
        FROM export_info x3
        WHERE x2.product = x3.product
          AND x1.country = x3.country))
```

where the database table `export_info(country, product)` is assumed. The SQL query above corresponds to the paraphrased question: '*Is there a country, for which there is no product not exported by this country?*'

The question is answered more naturally with the following (simplified) SQL+ query:

```
temprel(t1, (country, card)
       'SELECT x3.country, COUNT(DISTINCT x3.product)
        FROM export.info x3
        GROUP BY x3.country').
temprel(t2, (card)
       'SELECT COUNT(DISTINCT x4.country)
        FROM t1 x4
        WHERE x4.card= (SELECT COUNT(DISTINCT x2.product)
                        FROM export_info x2)').
query(yesno, noreverse,
      'SELECT *
       FROM t2 x5
       WHERE x5.card = (SELECT COUNT(DISTINCT x1.country)
                        FROM export_info.x1)').
```

The first `temprel` creates a temporary table `t1(country,card)` containing the number of products exported by each country. The second `temprel` counts the tuples in `t1` corresponding to countries exporting all products, and stores the result in t2. Finally, the `query` checks if the number saved in `t2` is equal to the number of existing countries.

Although temporary relations may help clarify the meaning of SQL queries, the clarity of the generated SQL code was not considered important in the case of this project. The temporary relation strategy is also inefficient, since it creates new temporary tables during the evaluation. So, this strategy is not used in Masque/SQL.

4.5 Accessing the relational database through set predicates

The extended Masque/PRO evaluation strategy was argued to be inefficient, because it uses Prolog's depth-first, backtracking strategy to join partial results retrieved by the DBMS. Instead, the evaluation methods of the previous section are more efficient, because they create an overall query, expressed in SQL, and let the DBMS find all the solutions. However, using the later methods, we lose the ability to use Prolog to complement the knowledge stored in the relational database.

The problem is that, although we want to be able to express the relational query using the Prolog formalism (i.e the relational query language must be 'logically tightly connected' to Prolog), we also want the relational query to be executed by the DBMS, not by Prolog (i.e the relational query language must be 'physically loosely connected' to Prolog).

[Draxler 92] proposes a method to achieve such a 'logically tight and physically loose' coupling between Prolog and relational databases. The terms 'physically loose' and 'logically tight' are borrowed from Draxler. Also consult [Draxler 92] and [Ceri *et al* 90] for an extensive discussion of the various methods for the connection of Prolog and relational databases.

The method proposed in [Draxler 92] uses an extension of Prolog's 'set' predicates (`setof`, `bagof`, `findall`) to mark subparts of the Prolog query that need to be evaluated by the DBMS. The special predicate

```
dbsetof(ProjectionTerm, DatabaseGoal, ResultSet)
```

is similar to Prolog's ordinary `setof`, except that the `ResultSet` is created by the DBMS, not the Prolog interpreter.

Consider again the example that was used to demonstrate the inefficiency of the extended Masque/PRO strategy:

```
answer([_1]):- country(_1),
               european(_1),
               borders(_1, greece)
```

Using `dbsetof` the query could be expressed as:

```
dbsetof(Country, (country(Country),
                  european(Country),
                  borders(Country, greece))
        Answer_list).
```

where `country/1`, `european/1`, and `borders/2` are 'linked' to database tables, as in the extended Masque/PRO strategy of section 4.3.

Note that in this case the conjunction list:

```
(country(Country),european(Country),borders(Country, greece))
```

is evaluated by the DBMS. The conjunction list is translated into a suitable overall query towards the relational database, and the RDBMS is left to find the answers.

[Draxler 92] provides a sample implementation of `dbsetof`, and of the similar `dbbagof`, `dbfindall`, that generates the expected results by translating the second-argument goal into an SQL query which is executed against the relational database.

Since we are allowed to mix `dbsetof` instances with ordinary Prolog predicates, the full power of Prolog is still available. Thus transitive closure questions could be answered as described in section 4.3.

The advantage of this approach is that, without sacrificing the expressiveness of Prolog, predicate instances referring to the relational database can be grouped together and translated into a single relational query. However, non-trivial modifications to the Masque kernel would be needed, in order to ensure that all predicate instances referring to the relational database are grouped together within `dbsetof` instances, in the LQL queries. Also, the extended set predicate implementations were not available at the beginning of this project. Hence, although less powerful, an evaluation method that translates LQL queries into SQL was still considered more suitable to the purposes of this project.

4.6 Summary

In this chapter, several methods for the evaluation of logical queries against relational databases were discussed. Using a Prolog-to-relational database interface of the type described by Lucas would be the easiest way to connect Masque to a relational database, since no change to the Masque code would be needed. However, the resulting system could be considerably inefficient.

A more efficient approach is to translate each LQL query into a single SQL query, and execute the SQL code against the relational database. This is the method used in the Essex front-end and IBM's LanguageAccess. This method is also the most portable. No special code is needed to interface the Prolog interpreter to the relational database. However, this approach restricts the knowledge available during the question answering to the knowledge that can be retrieved from the relational database using a single SQL query. As it was shown, there are cases where complementary knowledge is needed.

A better approach would be to use Draxler's database set predicates. This way the logical query is treated again as a Prolog program, but the subparts of the logical query referring to the relational database are grouped together, and executed as a single relational query by the DBMS. This approach combines the power of the extended Masque/PRO method and the efficiency of the translation to SQL methods. However, non-trivial modifications to Masque's kernel would be needed, in order to use this approach. So, an LQL to SQL evaluation strategy was considered more suitable to the purposes of this project.

Chapter 5

Evaluating LQL Queries by Translating them into SQL

5.1 Introduction

In the previous chapter, several methods for the evaluation of logical queries against a relational database were discussed. Although each method had its own advantages, it was argued that the method of translating each logical query into a single SQL query was probably the most suitable to the purposes of this project.

This chapter contains a detailed description of an algorithm that can be used to translate each LQL query into a proper SQL query. The generated SQL query is proper in the sense that, when executed against a relational world database of a certain structure, it will produce results that can be interpreted to denote answers satisfying the LQL query.

The algorithm presented in this chapter is the one used in Masque/SQL. It has been extensively tested, and it efficiently produced suitable SQL code in all cases. However, this chapter contains neither a proof of the algorithm's correctness, nor a formal assessment of the algorithm's efficiency.

- The translation algorithm generates SQL queries that behave correctly when executed against a relational world database of a certain structure. The second section of this chapter describes the assumed form of the world database.

- The third section discusses the steps the knowledge engineer needs to take, in order to logically connect the world database to LQL's domain-dependent predicates. This is a necessary step before the translation algorithm can be used.

- The fourth section provides a detailed description of the translation algorithm. Along with the algorithm description, notes explaining how to interpret the results of the generated SQL queries are also given.

- The fifth section discusses some cases where the translation algorithm cannot be used to evaluate LQL queries.

- Finally, the sixth section summarises the chapter.

Step-by-step LQL to SQL translation examples, using this chapter's algorithm, can be found in Appendix B.

5.2 The form of the world-database

This section describes the assumed form of the commercial relational database, acting as the world-database.

The following discussion assumes that the world-database is used to evaluate LQL queries, with respect to an LQL interpretation I (see section 3.4.1). The notation f/ar denotes a predicate with functor f and arity ar.

The world-database contains information about world entity-sets (i.e. unary entity-relations) and entity-relations.

For every LQL type-A domain-dependent predicate $func/n$, iff $I(func/n) = erel$, then the world entity-relation $erel$ is represented in the database as a relation $func\text{-}n$, created as[1]:

---

[1]Ingres doesn't allow table names to contain the character '-'; so the character '#' is used instead in Masque/SQL.

```
CREATE TABLE func-n
(ARG1 type_1
 ARG2 type_2
     ...
 ARGN type_N)
```

such that: $\forall\,(entity_1, \ldots, entity_n)\ \forall\,(atom_1, \ldots, atom_n)$

$$(entity_1, \ldots, entity_n)\,\epsilon\, erel \land$$
$$Eval_I(atom_1) = entity_1 \land \ldots \land Eval_I(atom_n) = entity_n$$
$$\rightarrow\ (atom_1, \ldots, atom_n)\,\epsilon\, func\text{-n}$$

$type_1, type_2, \ldots, type_N$ must be suitable SQL datatypes (such as `VARCHAR`, `FLOAT`). Some care is needed when selecting the SQL datatypes of the database tables, since some of the SQL functions (such as `max`, `av`) that often appear in the generated SQL code only work for numeric datatypes. The current Masque/SQL version allows string (e.g. `VARCHAR`, `CHAR` in Ingres) and numeric (e.g. `FLOAT`, `INTEGER` in Ingres) datatypes only.

For example,

- If $erel = \{(Greece, Greek), (UK, English), (USA, English)\}$

  is a world entity-relation

- and:

$$\begin{aligned} Eval_I(greece) &= Greece \\ Eval_I(greek) &= Greek \\ Eval_I(uk) &= UK \\ Eval_I(english) &= English \\ Eval_I(usa) &= USA \end{aligned}$$

- and $spoken\_in/2$ is a type-A domain-dependent predicate

- and $I(spoken\_in/2) = erel$

  then the world-database will contain a relation of the form:

spoken_in-2	
ARG1	ARG2
greece	greek
uk	english
usa	english

5.3 The interface to the world-database

Before the translation algorithm can be used, the knowledge-engineer has to logically 'connect' the world database to LQL's domain-dependent predicates.

- For every type-A domain-dependent predicate, a suitable database relation must be created, as described in the previous section. This relation may have the form of an SQL *view*, if we need to use an existing database, the tables of which don't have the appropriate structure.

- For every type-B domain-dependent predicate (i.e. introduced by a noun or adjective acting over a pair-set), the meaning of the predicate must be expressed as an SQL query of the form:

  ```
  SELECT ...
  FROM PAIR_SET, rest_from_expression
  WHERE ...
  ```

  The select query must produce a one-attribute relation. `PAIR_SET` is an imaginary relation over the relation schema (`FIRST`, `SECOND`), representing the pair-set the noun/adjective acts over. *rest_from_expression* must not contain `PAIR_SET`.

  This method restricts the type-B domain-dependent predicates that can be introduced to those that can be expressed as an SQL query of this form. Although one can think of type-B predicates that cannot be expressed this way, it is not difficult to express most practically useful type-B predicates.

  Consider the example of a type-B domain-dependent predicate `has50`, acting over LQL pair-sets of the form:

  ```
  [70:person1, 50:person2, 57:person3, 50:person4]
  ```

  We want `has50` to 'pick' persons associated with the number 50; i.e. we want

  ```
  has50(person2,[70:person1,50:person2,57:person3,50:person4])
  has50(person4,[70:person1,50:person2,57:person3,50:person4])
  ```

to be both true, but

```
has50(person3,[70:person1,50:person2,57:person3,50:person4])
```

to be false.

The knowledge engineer could define the meaning of `has50` as:

```
SELECT SECOND
FROM PAIR_SET
WHERE FIRST = 50
```

5.4 The translation algorithm

This section presents an algorithm for translating each LQL query to an SQL query. When the generated SQL query is executed against the world-database, it produces results that can be interpreted to denote answers that satisfy the LQL query.

In the following sections, $trans(e)$ denotes the translation of e, where e is a fragment of an LQL query.

Along with the description of the translation algorithm, notes explaining how to interpret the results of the generated SQL queries will also be given.

5.4.1 The bindings structure

The translation algorithm that will be described uses a global data structure, which I will call the *bindings structure* (BS), to store bindings corresponding to LQL variables.

The bindings structure has the logical form:

Variable	Binding
$variable_1$	$binding_1$
$variable_2$	$binding_2$
$variable_3$	$binding_3$
...	...

where $variable_1$, $variable_2$, ... are LQL variables (e.g. _10512, _23), and $binding_1$, $binding_2$, ... are fragments of SQL code.

The bindings structure is cleared whenever the translation of a new LQL query begins. A variable cannot have more than one binding in the BS.

In the following, $bndg(var)$ will denote the binding in the BS, corresponding to var.

5.4.2 Type-A domain-dependent predicate instances

A type-A domain-dependent predicate instance of the form (the asterisk denotes that the corresponding argument can be an LQL variable):

$$functor(atom_1^*, \ldots, atom_N^*)$$

is translated into the SQL query:

```
SELECT *
FROM functor-N corr_name
WHERE condition_1
  AND condition_2
      ...
  AND condition_M
```

where $M \leq N$ and $corr\_name$ is generated by a call to a correlation-names generator. A correlation name can be thought as a relation alias, or a tuple variable if we treat SQL expressions as Tuple Relational Calculus expressions [Ullman 88]. It is useful when an SQL query uses many instances of the same relation [ING90b]. Each call to the correlation-names generator returns a unique string of the form $\mathtt{REL}_i$ (e.g. $\mathtt{REL}_1, \mathtt{REL}_2, \ldots$).

The conditions are generated as follows:

For each argument $atom_i^*$ of the predicate instance,

- if $atom_i^*$ is an LQL variable that has no binding in the BS, then no condition is generated, but the pair $(atom_i^*, corr\_name.\mathtt{ARG}_i)$ is inserted into the BS.

- if $atom_i^*$ is an LQL variable that has a binding in the BS, then:

- if $bndg(atom_i^*)$ is a fragment of SQL code starting with the keyword `SELECT`, then the condition $corr\_name.$`ARG`$_i$ `IN` $(bndg(atom_i^*))$ is generated.
- otherwise, the condition $corr\_name.$`ARG`$_i = bndg(atom_i^*)$ is generated.

- if $atom_i^*$ is not an LQL variable, then the condition $corr\_name.$`ARG`$_i = atom_i^*$ is generated.

E.g: The predicate instance $links(\_1285, athens, heraklion)$, where _1285 has no binding, would be translated into:

```
SELECT *
FROM links-3 REL1
WHERE REL1.ARG2 = 'athens'
  AND REL1.ARG3 = 'heraklion'
```

and the binding (_1285, `REL1.ARG1`) would be inserted into the BS.

Notice that '`athens`' and '`heraklion`' need to be quoted in the generated SQL code. An implementation of the translation algorithm can easily check the datatypes of the corresponding table attributes, to decide whether an atom needs to be quoted or not.

The predicate instance $stops(\_1285, nayplion)$ would then be translated into:

```
SELECT *
FROM stops-2 REL2
WHERE REL2.ARG1 = REL1.ARG1
  AND REL2.ARG2 = 'nayplion'
```

The reader will have noticed that the intermediate results of the translation algorithm may not be well-formed SQL queries.

5.4.3 Conjunction lists

Conjunction lists are translated using the *join strategy* of the previous chapter (see section 4.4.2).

A conjunction-list $P_1, \ldots, P_N$ is translated as follows:

- First we translate P_1, then P_2, etc. (The reader will recall that the contents of the BS are cleared only when the translation of a new LQL query begins.)

- We then form the set: $T = \{trans(P_1), \ldots, trans(P_N)\} \backslash \{e\}$
  'e' denotes the empty string. Some fragments of LQL queries are translated into the empty string.

- T will have the form:

  {SELECT $select\_expression_1$
  FROM $table_{1.1}\ corr_{1.1}$, $table_{1.2}\ corr_{1.2}$ ...
  WHERE $conditions_1$,

  SELECT $select\_expression_2$
  FROM $table_{2.1}\ corr_{2.1}$, $table_{2.2}\ corr_{2.2}$ ...
  WHERE $conditions_2$,

  ...,

  SELECT $select\_expression_M$
  FROM $table_{M.1}\ corr_{M.1}$, $table_{M.2}\ corr_{M.2}$ ...
  WHERE $conditions_M$}

  where $M \leq N$.

- The translation of the conjunction-list is:

  SELECT *
  FROM $table_{1.1}\ corr_{1.1}$, $table_{1.2}\ corr_{1.2}$ ...,
  $table_{2.1}\ corr_{2.1}$, $table_{2.2}\ corr_{2.2}$ ...,
  ...,
  $table_{M.1}\ corr_{M.1}$, $table_{M.2}\ corr_{M.2}$ ...
  WHERE ($conditions_1$)
  AND ($conditions_2$)
  ...
  ANd ($conditions_M$)

For example, the translation of the conjunction-list:

```
country(_9104),
official_language(_9104,_9103),
american(_9104)
```

is carried out as follows:

- `country(_9104)` is translated into:

```
SELECT *
FROM country-1 REL1
```

  and the binding (_9104,`REL1.ARG1`) is inserted into the BS.

- `official_language(_9104,_9103)` is translated into:

```
SELECT *
FROM official_language-2 REL2
WHERE REL2.ARG1 = REL1.ARG1
```

  and the binding (_9103,`REL2.ARG2`) is inserted into the BS.

- `american(_9104)` is translated into:

```
SELECT *
FROM american-1 REL3
WHERE REL3.ARG1 = REL1.ARG1
```

- the conjunction-list is translated into:

```
SELECT *
FROM country REL1, official_language REL2, american REL3
WHERE (REL2.ARG1 = REL1.ARG1)
  AND (REL3.ARG1 = REL1.ARG1)
```

5.4.4 Quantified conjunction lists

The translation of a quantified conjunction-list is the same as the translation of its body, i.e:

$$trans(var_1^\wedge \ldots ^\wedge var_n^\wedge(body)) = trans(body)$$

and $trans((body)) = trans(body)$

5.4.5 'setof' instances

The translation of a `setof` instance is the empty string. However, the translation has side-effects that affect the BS.

The translation of an instance $setof(var_1, quant\_conj\_list, var_3)$ is carried out as follows:

- First we keep a copy of the current state of BS; let BS' be the copy.

- We then translate the *quant_conj_list*. The result of the translation will have the form:

  SELECT *select_expression*
  FROM *from_expression*
  WHERE *where_expression*

  and BS will have possibly been altered.

- The SQL query:

  SELECT DISTINCT $bndg(var_1)$
  FROM *from_expression*
  WHERE *where_expression*

  is inserted into *BS'* as the binding of var_3, and BS' becomes the new bindings structure. Intuitively, as in the case of Prolog, any binding of var_1 or of variables appearing only within *quant_conj_list*, created during the translation of the *setof* instance, must not be 'exported' outside the *setof* instance. This is why the BS copy is used.

The translation of an instance $setof(var_1 : var_2, quant\_cond\_list, var_3)$ is similar, except that the binding of var_3 is now:

SELECT DISTINCT $bndg(var_1)$, $bndg(var_2)$
FROM *from_expression*
WHERE *where_expression*

The translation of an instance $setof(var_1 : [var_2], quant\_cond\_list, var_3)$ is exactly the same as the translation of $setof(var_1 : var_2, quant\_cond\_list, var_3)$.

For example, the translation of the `setof` instance:

```
setof(_9103:_9104,
      (country(_9104),
       official_language(_9104,_9103),
       american(_9104))
      _9102)
```

is carried out as follows:

- A copy of the BS is kept, let BS' be the copy.

- The conjunction-list is translated into:

```
SELECT *
FROM country REL1, official_language REL2, american REL3
WHERE (REL2.ARG1 = REL1.ARG1)
  AND (REL3.ARG1 = REL1.ARG1)
```

  and the bindings (_9104, `REL1.ARG1`), (_9103, `REL2.ARG2`) are inserted into the BS.

- The query:

```
SELECT DISTINCT REL2.ARG2, REL1.ARG1
FROM country REL1, official_language REL2, american REL3
WHERE (REL2.ARG1 = REL1.ARG1)
  AND (REL3.ARG1 = REL1.ARG1)
```

  is inserted into *BS'* as the binding of _9102, and BS' becomes the new bindings structure.

5.4.6 Type-B domain-dependent predicate instances

A type-B domain-dependent predicate instance

$$functor(atom^*, variable)$$

the meaning of which is expressed in SQL by the query:

SELECT $attr\_expression$
FROM PAIR_SET, $rest\_from\_expression$
WHERE $where\_expression$

and where $bndg(variable) =$[2]

SELECT DISTINCT $attr_1$, $attr_2$
FROM $arg2\_from\_expression$
WHERE $arg2\_where\_expression$

is translated as follows:

[2]The restrictions of section 3.3.9 guarantee that *variable* will already have a binding in the BS, when the translation of the type-B instance begins.

- if $atom^*$ is an LQL variable without a binding in BS, then the predicate instance is translated into the empty string (e). However, as a side-effect,

  SELECT $new\_attr\_expression$
  FROM $arg2\_from\_expression$, $rest\_from\_expression$
  WHERE ($arg2\_where\_expression$)
    AND ($new\_where\_expression$)

  becomes the binding of $atom^*$ in the BS, where:

  1. $new\_attr\_expression$ is produced from $attr\_expression$, by replacing all occurrences of FIRST, SECOND by $attr_1$, $attr_2$ respectively.

  2. $new\_where\_expression$ is produced from $where\_expression$, by replacing all occurrences of FIRST, SECOND by $attr_1$, $attr_2$ respectively, and by recursively replacing any subselect query of the form:

     (SELECT $subselect\_attr\_expression$
      FROM PAIR_SET, $subselect\_rest\_from\_expression$
      WHERE $subselect\_where\_expression$)

     that appears within $where\_expression$ by:

     SELECT $new\_subselect\_attr\_expression$
     FROM $arg2\_from\_expression$, $subselect\_rest\_from\_expression$
     WHERE ($arg2\_where\_expression$)
       AND ($new\_subselect\_where\_expression$)

     $new\_subselect\_attr\_expression$, $new\_subselect\_where\_expression$ are defined similarly to $new\_where\_expression$, $new\_attr\_expression$.

  For example, consider a type-B predicate $no\_of\_english/2$, acting over sets of the form:

  $$[english : uk, english : usa, greek : greece, english : australia]$$

  which is supposed to report the number of countries where English is the official language.

  The meaning of $no\_of\_english/2$ in SQL is:

```
SELECT count(SECOND)
FROM PAIR_SET
WHERE FIRST = 'english'
```

A predicate instance of the form $no\_of\_english(\_1125, \_1672)$, where _1125 has no binding in BS, and $bndg(\_1672)$ =

```
SELECT DISTINCT REL1.ARG1, REL1.ARG2
FROM language_spoken-3 REL1
WHERE REL1.ARG3 = 'america'
```

would have added the binding of _1125:

```
SELECT count(REL1.ARG2)
FROM language_spoken-3 REL1
WHERE (REL1.ARG3 = 'america')
  AND (REL1.ARG1 = 'english')
```

in the BS.

- if $atom^*$ is not an LQL variable (let $nonvar = atom^*$ in this case), or if it is an LQL variable with a binding in BS (let $nonvar = bndg(atom^*)$ in this case), then the predicate instance is translated into:

```
SELECT
FROM
WHERE nonvar IN (SELECT new_attr_expression
                 FROM arg2_from_expression, rest_from_expression
                 WHERE (arg2_where_expression)
                   AND (new_where_expression)
```

where all *expressions* are defined as above.

For example, a predicate instance of the form $no\_of\_english(3, \_1672)$, where $bndg(\_1672)$ is as above, would have been translated into:

```
SELECT
FROM
WHERE 3 IN (SELECT count(REL1.ARG2)
            FROM language_spoken-3 REL1
            WHERE (REL1.ARG3 = 'america')
              AND (REL1.ARG1 = 'english'))
```

5.4.7 \+ instances

\+ instances are translated using the *negation strategy* discussed in the previous chapter (see section 4.4.2).

An instance \+(*conjunction_list*) is translated as follows:

- First, a copy of the current BS is kept, let BS' be the copy.
- Then, *conjunction_list* is translated into *trans*(*conjunction_list*).
- BS' becomes the bindings structure. Intuitively, as in the case of Prolog, any bindings created when processing the negated conjunction list must not be 'exported' outside the negation instance. This is why the BS copy is used.
- The translation of \+(*conjunction_list*) is:

  ```
  SELECT
  FROM
  WHERE NOT EXISTS (trans(conjunction_list))
  ```

For example, the translation of:

```
\+(country(_9104),
   official_language(_9104,_9103),
   american(_9104))
```

is carried out as follows:

- A copy of the BS is kept. Let BS' be the copy.
- The conjunction-list is translated into:

  ```
  SELECT *
  FROM country REL1, official_language REL2, american REL3
  WHERE (REL2.ARG1 = REL1.ARG1)
    AND (REL3.ARG1 = REL1.ARG1)
  ```

- The \+ instance is translated into:

  ```
  SELECT
  FROM
  WHERE NOT EXISTS (SELECT *
                    FROM country REL1,
                         official_language REL2,
                         american REL3
                    WHERE (REL2.ARG1 = REL1.ARG1)
                      AND (REL3.ARG1 = REL1.ARG1))
  ```

  and the BS is restored to BS'.

5.4.8 Other built-in predicates

The LQL specification of chapter 3 defines a built-in predicate $is(X, Y)$. is instances can be easily eliminated by substituting any X in the LQL query by Y. It is assumed that all is instances have been eliminated before the algorithm is used.

- $trans(true) = e$.

- $length$ instances of the form $length(set\_variable, unsgn\_int^*)$ are translated as follows:

  - $set\_variable$ will have a binding of the form:

    SELECT $select\_expression$
    FROM $from\_expression$
    WHERE $where\_expression$

    The restrictions of section 3.3.9 guarantee that $set\_variable$ will always have a binding in BS, when the translation of an instance of this type begins.

  - If $unsgn\_int^*$ is an LQL variable with no binding in BS, then the instance is translated into e, but as a side-effect:

    SELECT count(∗)
    FROM $from\_expression$
    WHERE $where\_expression$

    is inserted into the BS as the binding of $unsgn\_int^*$.

  - If $unsgn\_int^*$ is not an LQL variable (let $nonvar = unsgn\_int^*$ in this case), or if $unsgn\_int^*$ is an LQL variable with a binding in BS (let $nonvar = bndg(unsgn\_int^*)$ in this case), then the instance is translated into:

    SELECT
    FROM
    WHERE $nonvar$ = (SELECT count(∗)
    FROM $from\_expression$
    WHERE $where\_expression$)

- Instances of min are translated as if min were a type-B domain dependent predicate, defined as:

```
SELECT DISTINCT SECOND
FROM PAIR_SET
WHERE FIRST = (SELECT min(FIRST)
               FROM PAIR_SET)
```

- Instances of *max* are translated similarly.

- Similarly, *tot* and *av* behave as if they were defined as:

  ```
  SELECT sum(FIRST)
  FROM PAIR_SET
  ```

  and:

  ```
  SELECT avg(FIRST)
  FROM PAIR_SET
  ```

  respectively.

- Predicate instances of the form $variable < number$ are translated into:

  SELECT
  FROM
  WHERE $bndg(variable) < number$

- Predicate instances of the form $variable > number$ are translated similarly.

- Predicate instances of the form $greater\_than(var_1, var_2)$ are translated into:

  SELECT
  FROM
  WHERE $bndg(var_1) > bndg(var_2)$

- Predicate instances of the form $less\_than(var_1, var_2)$ are translated similarly.

5.4.9 LQL queries

In the following paragraphs the square brackets ([]) are used to denote SQL code that may not be present in some cases.

1. yes/no queries: LQL queries of the form:

$$answer([]) : -conjunction\_list$$

where $trans(conjunction\_list) =$

SELECT *select_expression*
FROM *from_expression*
[WHERE *where_expression*]

are translated into:

SELECT *not_empty_select_expression*
FROM *not_empty_from_expression*
[WHERE *where_expression*]

where *not_empty_select_expression* is:

- *select_expression*, if $select\_expression \neq e$
- '*', if $select\_expression = e$

and *not_empty_from_expression* is:

- the same as *from_expression*, if $from\_expression \neq e$
- `'dummy_table'`, if $from\_expression = e$, where `dummy_table` is a table with exactly one row.

If the resulting SQL query retrieves no rows, then the LQL query evaluates to *false*; otherwise it evaluates to *true*.

2. **simple enumeration queries:** LQL queries of the form:

$$answer([variable]) : -conjunction\_list$$

are translated as follows:

- First *conjunction_list* is translated. $trans(conjunction\_list)$ will have the form:

  SELECT *select_expression*
  FROM *from_expression*
  [WHERE *where_expression*]

- If, after the translation of *conjunction_list*, *variable* has no binding in the BS, then the translation process stops, and the LQL query evaluates to *false*.

- If, after the translation of *conjunction_list*, *bndg*(*variable*) has the form:

  SELECT $variable\_select\_expression$
  FROM $variable\_from\_expression$
  [WHERE $variable\_where\_expression$]

  then the LQL query is translated into:

  SELECT $not\_empty\_variable\_select\_expression$
  FROM $not\_empty\_variable\_from\_expression$
  [WHERE $variable\_where\_expression$]

  where the *not_empty* expressions are defined similarly as in case (1).

- In any other case, the LQL query is translated into:

  SELECT $bndg(variable)$
  FROM $not\_empty\_from\_expression$
  [WHERE $where\_expression$]

The resulting SQL query is executed; if the SQL query retrieves no rows, then the LQL query evaluates to $false$; otherwise the results of the evaluation are described by the results of the SQL query.

3. complex enumeration queries: LQL queries of the form:

$$answer([var_1, var_2]) : -conjunction\_list$$

are translated as follows:

- First, *conjunction_list* is translated. *trans*(*conjunction_list*) will have the form:

  SELECT $select\_expression$
  FROM $from\_expression$
  [WHERE $where\_expression$]

- If, after the translation of *conjunction_list*, var_1 or var_2 has no binding in the BS, then the translation process stops, and the LQL query evaluates to $false$.

- If, after the translation of *conjunction_list*, $bndg(var_2)$ is *not* a fraction of SQL code starting with the keyword SELECT, then the LQL query is translated into:

  SELECT $bndg(var_1), bndg(var_2)$
  FROM $not\_empty\_from\_expression$
  [WHERE $where\_expression$]

  where $not\_empty\_from\_expression$ is defined as in case (1).

- Otherwise, $bndg(var_2)$ will have the form:

  SELECT $var_2\_select\_expression$
  FROM $var_2\_from\_expression$
  [WHERE $var_2\_where\_expression$]

  - If: $var_2\_select\_expression = aggregate(\ldots)$

    where *aggregate* can be one of: **min, max, count, avg, sum**

    then the LQL query is translated into:

    SELECT $bndg(var_1)$, $aggregate(\ldots)$
    FROM $union(not\_empty\_from\_expression,\ var_2\_from\_expression)$
    [WHERE $var_2\_where\_expression$]
    GROUP BY $bndg(var_1)$

    where $not\_empty\_from\_expression$ is defined as in case (1), and *union* denotes concatenation with duplicates removed.

  - Otherwise, the LQL query is translated into:

    SELECT $bndg(var_1)$, $var_2\_select\_expression$
    FROM $union(not\_empty\_from\_expression,\ var_2\_from\_expression)$
    [WHERE $var_2\_where\_expression$]

The resulting SQL query is executed; if the SQL query retrieves no rows, then the LQL query evaluates to *false*; otherwise the results of the evaluation are described by the results of the SQL query.

4. **cardinality queries:** LQL queries of the form:

$$answer([var_1\#var_2]) : -conjunction\_list$$

are translated as follows:

- First $conjunction\_list$ is translated. The result of the translation is ignored.
- If, after the translation of $conjunction\_list$, var_1 or var_2 has no binding in the BS, then the translation process stops, and the LQL query evaluates to *false*.
- Otherwise, var_1 will have a binding of the form:

  SELECT $var_1\_select\_expression$
  FROM $var_1\_from\_expression$
  [WHERE $var_1\_where\_expression$]

and the LQL query is translated into:

SELECT $not\_empty\_var_1\_select\_expression$
FROM $not\_empty\_var_1\_from\_expression$
[WHERE $var_1\_where\_expression$]

Where the *not_empty* expressions are defined similarly as in case (1).

The SQL query $bndg(var_2)$ could also be executed, to help the system answer follow-up questions (see comments about cardinality queries in section 3.3.10).

LQL to SQL translation examples, using the algorithm presented, are given in Appendix B.

5.5 Shortcomings of the Masque/SQL evaluation method

As mentioned in the previous chapter, there are some cases where a question against a relational database cannot be answered using a single SQL query. In section 4.3, one such question referring to a table's transitive closure was presented, and it was argued that the Essex system could not answer it, unless the transitive closure were to be stored explicitly in the database, because the transitive closure cannot be computed using a single SQL query.

Masque/SQL also attempts to answer the user's questions by generating a single SQL query for each question. It is therefore reasonable to expect that Masque/SQL will not be able to answer transitive closure questions.

Consider again the example of the chemical plant parts, presented in the previous chapter, and the user's question: '*What are the parts of engine1?*' Masque would generate an LQL query of the form:

```
answer([_10]):- part(_10),
                part_of(_10, engine1)
```

Using the algorithm of this chapter, the LQL query would have been translated into the SQL query:

```
SELECT REL1.ARG1
FROM PART-1 REL1, PART_OF-2 REL2
WHERE REL2.ARG1 = REL1.ARG1 AND
      REL2.ARG2 = ‘engine1‘
```

However, the SQL query could not be executed, because the knowledge engineer would not be able to define the necessary view `PART_OF-2`. `PART_OF-2` is the transitive closure of the table `IMMEDIATE_PART_OF-2`, and there is no way to define this view in SQL. The only way to answer the question would be to have `PART_OF-2` as a database table, but as it was explained in the previous chapter, this is problematic.

5.6 Summary

This chapter has provided a detailed description of an algorithm that can be used to translate any LQL query into a single SQL query. The SQL query generated by the algorithm is proper, in the sense that, when executed against a relational world database of a certain structure, it will produce results that can be interpreted to denote answers satisfying the LQL query.

Using this algorithm, any LQL query (with some exceptions) can be evaluated, by translating it into SQL and executing the SQL code against the relational world database.

Although neither a proof of the algorithm's correctness nor a formal assessment of its efficiency were provided in this chapter, the algorithm was implemented as part of Masque/SQL. It has been extensively tested, and has efficiently generated suitable SQL code in all cases.

Chapter 6

Implementing and Testing Masque/SQL

6.1 Introduction

This chapter describes Masque/SQL, the modified version of Masque produced during this project, that can be used as a front-end to relational databases.

- Masque/SQL adopts the logical query evaluation method of the previous chapter: an LQL to SQL translator has been implemented, that translates each logical query into a suitable SQL `select` command. The next section of this chapter presents the Masque/SQL architecture, focusing on the modules added or modified during this project.

- To test the modified system, a sample Masque/SQL domain called *geogsql* has been implemented. *geogsql* is a world geography domain, where all the geography data are kept in the external relational database. Section 3 describes *geogsql* and the tests that have been made with Masque/SQL.

- Section 4 summarises the chapter.

More detailed information about installing and using Masque/SQL can be found in [Androutsopoulos 92].

6.2 The Masque/SQL architecture

Figure 6–1 shows the architecture of Masque/SQL. From the modules shown, the *parser*, the *semantic interpreter*, the *scoper*, the structure of the two *dictionaries* and the *type hierarchy* remain as in Masque/PRO (see figure 2–1).

The *logical query simplifier* slightly simplifies the logical queries produced by the *scoper*. The simplifications made by the *logical query simplifier* are very shallow. For example, redundant existential quantifiers are removed, and `is(X,Y)` instances are eliminated, by substituting each `X` in the logical query by a `Y`. The simplifier could be improved, to remove *logically* redundant predicate instances from the LQL queries (see section 7.2). The *logical query simplifier* also makes some minor changes to the logical queries, to ensure that they fit exactly the LQL specification of chapter 3 (for example `must_satisfy` instances are removed).

The *LQL to SQL translator* is a Prolog implementation of the translation algorithm described in the previous chapter.

The *interface with data* contains Prolog facts that help the translator turn domain-dependent predicate instances into SQL code.

For each type-A domain-dependent predicate, a suitable declaration must be present in the *interface with data*. Each type-A declaration has the form:

```
type_A(Functor, List_of_argument_datatypes).
```

The datatypes (not to be confused with the *semantic* types of the type hierarchy) are used to determine whether an atom present in the corresponding argument position should be quoted in the SQL code or not. The possible datatypes are `number` and `string`.

For example, a type-A domain-dependent predicate:

`population_of`(*place*, *population*)

could be declared in the *interface with data* as:

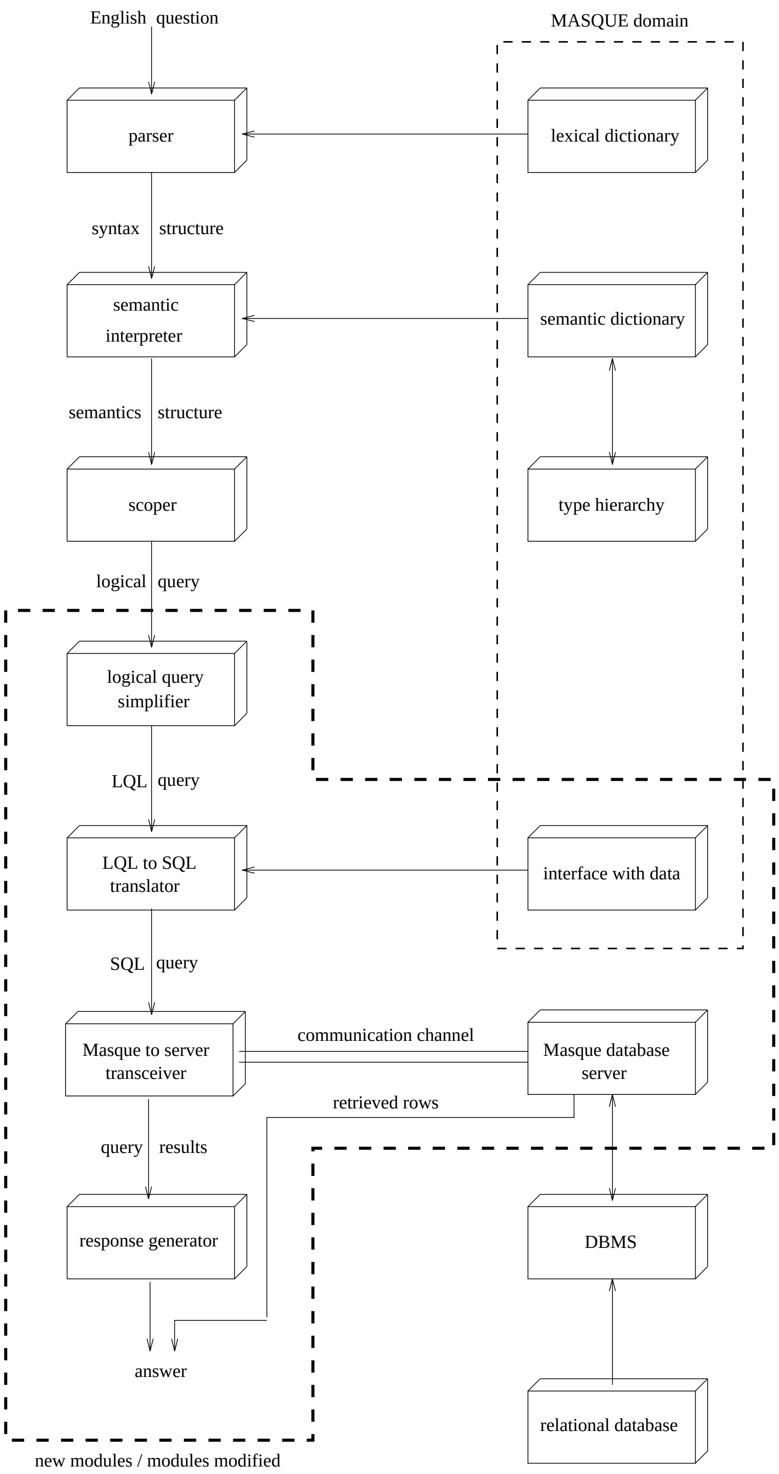

Figure 6–1: The Masque/SQL architecture

```
type_A(population_of, [string, number]).
```

A type-A declaration also implies that the corresponding table or view is present in the external relational database, as explained in section 5.3.

The meaning of each type-B domain-dependent predicate must be declared in the *interface with data*, in terms of a SQL select (see section 5.3). For example, we might want to define a type-B domain-dependent predicate, that would report the second parts of the pairs in a pair-set, having 50 as their first part. The SQL meaning of such a predicate is:

```
SELECT SECOND
FROM PAIR_SET
WHERE FIRST = 50
```

In the current version of Masque/SQL, the SQL meaning must be declared using a pseudo-SQL Prolog term, described in [Androutsopoulos 92]. In the above example, the *interface with data* would contain the definition:

```
type_B(has50, FIRST^SECOND^select([SECOND],
                                  [pair_set],
                                  [FIRST, = , 50])).
```

The generated SQL query is sent to the Masque database server. The server, implemented in C with embedded dynamic SQL, sends the appropriate messages to the DBMS of the external database, so that the SQL query can be executed. The current Masque/SQL version uses Ingres v.6.3 as its external database.

The database server runs as a separate process, in parallel with the main Masque process. This is done to bypass the complexity of linking the C code of the database server to the Masque SICStus Prolog code. The linking is difficult, because SICStus uses the UCB universe UNIX libraries, while Ingres uses the AT&T universe libraries. This architecture also allows Masque and the DBMS to execute on remote machines.

The two processes communicate with each other through the *communication channel*. The current Masque/SQL version uses a rather primitive file-based communication channel, that restricts the Masque main process and the database server to run on machines with a common file system. A better channel implementation could use UNIX sockets (see section 7.2).

The main Masque process sends SQL `select` commands to the server, along with a flag, signaling whether rows retrieved by the `select` are to be printed or not. In the case of *yes/no* queries, the server is instructed not to print any retrieved rows, since we are only interested to know if the `select` retrieves at least one row or not. In all other cases, the server is instructed to print the retrieved rows on the screen.

After executing the SQL `select` (and possibly printing the retrieved rows), the database server reports an overall status flag to the *response generator*, showing if the `select` retrieved any rows or not. The response generator uses this overall flag to complement the answer (possibly) printed by the database server, and to print suitable messages in cases of questions that cannot be answered.

The fact that any retrieved rows are printed directly by the database server, means that the pretty-print routines of the Masque/PRO response generator cannot be used. Thus, Masque/SQL might print results like `united_kingdom`, `4e7`, instead of `United Kingdom`, `40 million`. (See further improvements in section 7.2.)

The Masque/SQL response generator is less sophisticated than the Masque/PRO one. Although it will still print suitable messages to show at what level (unknown word, parse error, semantics inconsistency etc) the question answering failed, and it will allow some kinds of focusing questions[1], the Masque/SQL response generator is much less informative in cases where a question cannot be answered because the database doesn't contain any suitable data.

Consider for example the question: '*what is the largest continent?*'

Masque/PRO (using the 'geog80' geography domain) would generate the logical query:

[1]E.g. if the user asks the yes/no question '*are there any continents?*', Masque/SQL will answer '`Yes, do you want to know more?`'. If the user replies '*yes*', Masque/SQL will automatically rephrase the first question to '*what are the continents?*' and print the new results.

```
answer([_8397]):- setof(_8400:_8401,
                        (continent(_8401),
                         must_satisfy(has_area(_8401,_8400)),
                        _8399),
                  max(_8397,_8399)
```

and would report:

`'I do not know the value of has_area applied to any continent.'`

because it only knows the areas of the countries, not the areas of the continents.

In Masque/PRO the logical query is executed by the Prolog interpreter. When the interpreter calls `has_area(_8401, _8400)`, the goal fails. Masque/PRO keeps an internal 'ignorance note' for each failed `must_satisfy` goal; these notes help the system explain why the question couldn't be answered.

Instead, in Masque/SQL the logical query is translated into a single SQL `select` which is executed as a whole. If the `select` retrieves no rows, Masque/SQL can only report '`no answers found`', since it doesn't know which particular subparts of the SQL `select` failed, and which of the failed subparts shouldn't have failed. The Masque/SQL response generator could, however, be improved. See section 7.2.

6.3 Testing Masque/SQL with the 'geogsql' domain

To test Masque/SQL, a sample world geography domain, called *geogsql*, was created. The world geography is a good knowledge domain to test Masque/SQL, because:

- The geography concepts are familiar to everybody, and thus the users don't find it difficult to understand what sort of questions the system is supposed to answer.
- The relations between the various entities of a geography world are complex enough to allow linguistically interesting questions to be asked.

- A similar domain, called *geog80*, exists for Masque/PRO; this allows interesting comparisons to be made between the two systems. In *geog80* the geography data are stored in the Prolog database. The Ingres tables used by *geogsql* were created by exporting the geography data from *geog80*; thus both systems should 'know' the same facts.

The basic database tables used in *geogsql* are 10 in total. The following table shows the sizes of the database tables used:

TABLE	COLUMNS	ROWS
ocean	1	5
sea	1	6
country	8	156
city	3	71
borders	2	856
continent	1	6
river_in	3	129
is_in	2	796
special_latitude	2	5
in_continent	2	18

Table `is_in` implements a transitive closure using the brute-force method discussed in 4.3. Several views not shown in the table are also used, to 'convert' the structures of these basic database tables to the structures needed by the translation algorithm (see section 5.3).

The *lexical dictionary*, *the semantic dictionary* and the *type hierarchy* of *geogsql* are the same as the ones used in *geog80*[2]. This should allow exactly the same questions to be asked in both domains.

Indeed, Masque/SQL, with the *geogsql* domain loaded, was able to answer all the sample questions that accompany the Masque/PRO code, although, as explained in the previous section, the Masque/SQL responses can be less informative in some cases.

[2]The only difference is that the *geogsql* semantic dictionary uses the built-in predicate `is(X,Y)`, instead of the domain-dependent `size(X,Y)`. In *geog80*, `size/2` was implemented in Prolog as `size(X,X)`.

Sample questions and the corresponding answers produced by Masque/SQL can be found in Appendix C. It is interesting to note that Masque/SQL answered the questions almost as fast as Masque/PRO. In almost all cases the answers were generated under 10 seconds (which includes the time used by the DBMS to retrieve the answers). The times given in Appendix C are CPU times. The actual response times are slightly bigger. The tests were made on a heavily loaded Sequent Symmetry machine of the Edinburgh University Computing Service. Also note that no SQL indices were used during the tests.

The efficiency of Masque/SQL is impressive enough, if one considers that the data used by Masque/PRO are stored in main memory, while Masque/SQL has to access the external relational database. It should be noted, however, that Masque/SQL is not as widely tested as Masque/PRO, and it may well be the case that Masque/SQL will not be able to handle questions in other domains so well. For example, in *geogsql* no type-B domain-dependent predicates are used, since the built-in predicates cover all the cases of nouns/adjectives acting over entity-pair sets.

6.4 Summary

Masque/SQL, the Masque version produced during this project, can be used as a front-end to commercial relational databases. Masque/SQL answers the user's questions by translating each logical query into an SQL `select` command. The translation algorithm of the previous chapter is used.

Masque/SQL has been tested with a world geography domain, similar to the *geog80* domain of Masque/PRO. The system was able to answer all the sample questions that accompany the Masque/PRO code, almost as fast as Masque/PRO.

Chapter 7

Conclusions and Further Improvements

This chapter summarises the conclusions of the project, and proposes further improvements to Masque/SQL.

7.1 Conclusions

This project has shown that Masque can be modified to become a powerful natural language front-end to relational databases.

The system was extended, rather than radically modified, by adding new modules that transform the Masque logical queries into appropriate SQL code, while the Masque kernel (the true linguistic front-end that generates the logical queries) remained unmodified. This approach has allowed the resulting system to maintain the full linguistic coverage and functionality of the original Masque, and has helped to complete the necessary modifications in a relatively short period of time.

The modifications made to the system were based on a formal description of the logical queries generated by Masque. Although these logical queries are actually Prolog expressions, and in the original system they were executed by the Prolog interpreter, it was shown that they can be treated as expressions of an intermediate meaning representation language, called LQL, which is much simpler than Prolog.

Particular care was taken to restrict the LQL syntax, so that it only allows the forms of the logical queries generated by Masque, rather than the full set of Prolog terms. Similarly, the semantics of LQL were kept as 'weak' as possible, dismissing elements of the Prolog semantics that unnecessarily restricted the possible logical query evaluation strategies.

Although the resulting meaning representation language is somehow ad hoc, especially when compared to the logical languages used by systems such as the Essex front-end, it was shown that it can be defined rigorously, and that it can be used to express the meaning of complex English questions. As the LanguageAccess example shows, very similar meaning representation languages are used by other natural language front-ends too.

Several possible methods for the evaluation of LQL queries against relational databases were discussed. An approach based on a general-purpose transparent Prolog-database-to-relational-database interface would be the easiest way to attach Masque to a relational database. However, the resulting system could be extremely inefficient, as the DBMS would only be used to retrieve partial results, and these partial results would be joined by the Prolog interpreter. Instead, an LQL to SQL translation approach was used, and it was shown that LQL queries can be efficiently translated into appropriate SQL queries.

The modified Masque system generates a single overall SQL query for each user question. The SQL query is transmitted to the relational database, and the DBMS is responsible to retrieve all the necessary data, using its own specialised planning and optimising techniques. This way the full power of the relational DBMS is available when answering the questions. Indeed, Masque/SQL answers complex English questions almost as fast as Masque/PRO. This architecture also allows Masque/SQL to be easily portable to any relational database that supports SQL.

However, the ad hoc nature of LQL reflects on the LQL to SQL translation algorithm, which is, as a result, equally ad hoc. It is thus very difficult to prove the correctness of the LQL to SQL transformation used, and to guarantee the validity of the SQL code generated. On the contrary, more principled logic languages, used in systems such as the Essex front-end, have convenient mathematical properties, that allow the correctness of each intermediate transformation to be provable.

There is also a more significant limitation of Masque/SQL: as with all front-ends that transform each question into a single SQL query, there is no way to complement the knowledge stored in the relational database by Prolog (or generally logic) rules. From this point of view, a Masque version based on a transparent Prolog-to-relational database interface, although less efficient, is more powerful.

A better approach would be to use Draxler's extended set-predicates, that combine the efficiency of the DBMS with the expressiveness of Prolog. However, the Masque kernel would have to be modified to use the extended set-predicates in the generated logical queries, and such a modification could probably not have been completed during this project.

Despite its limitations, Masque/SQL seems to be efficient and portable enough to be used in real-life applications. Also, the fact that it generates SQL code might make it easier for an MIS manager to appreciate it as a practical tool. The system can also be used as an assistant for database programmers that need to write complex SQL queries, or even as an SQL tutor.

The *geogsql* domain has helped to demonstrate that the new system can cope efficiently with complex English questions, as well as the original system could. However, a real-life domain, that could be used to answer questions against an existing corporate database, is still needed to test Masque/SQL in practice, and to check its viability as a practical tool.

7.2 Further Improvements

This section proposes several further improvements to Masque/SQL:

better channel: In the current Masque/SQL version, the main Masque process communicates with the Masque database server through a primitive file-based communication channel (see chapter 6). Both processes scan continuously the 'channel' files, waiting for a message to appear. This method wastes unnecessarily computer resources, and also restricts the two processes to run on machines sharing the same file-system. A better channel, based on UNIX `sockets` or RPC (Remote Process Communication) should be implemented. This would allow faster and more reliable communications between the two processes, and would also allow Masque and the DBMS to run on remote machines.

better results printing: The Masque/PRO response generator provides a mechanism similar to the proper names preprocessor (see section 3.5), that allows results retrieved from the database to be converted into more natural formats, before being reported to the user [Lindop 86]. In the current Masque/SQL version, however, any results retrieved from the relational database are printed directly by the database server, and only an overall flag, showing if the SQL `select` retrieved any rows or not, is returned to the response generator (see figure 6–1). Thus, the pretty-print routines of the response generator are bypassed, which causes results like `united_kingdom`, `4e7` to be printed, instead of `United Kingdom`, `40 million`. Instead, the results retrieved from the relational database should be transmitted back to the main Masque process, and the response generator should be left to report the results.

dimensioned numbers: Masque/PRO provides a mechanism to handle dimensioned numbers (e.g. '*10 Km*', '*5 Kg*'). Dimensioned numbers are stored in the Prolog database as Prolog terms of the form: `value--unit`. Dimensioned numbers in the user question can be easily converted into the `value--unit` internal format, using the proper names preprocessor. The response generator pretty-print routines also allow internally stored dimensioned numbers to be converted into more natural formats, before being reported to the user.

For example, using the proper names preprocessor, Masque/PRO could transform the user question *'is the area of France greater than 100 ksqmiles?'* into the logical query:

```
answer([]):- area(_10, france),
             greater_than(_10, 100--ksqmiles)
```

and, if given the question '*what is the area of France?*', Masque/PRO would respond '`212 ksqmiles`'.

Masque/SQL cannot cope with dimensioned numbers. The problem is that there is no convenient way to store values along with their dimensions, as attributes of a database relation.

In Masque/SQL all values are stored in the relational database as non-dimensioned numbers. As a result, if asked the area of France, Masque/SQL reports simply '`212000`'.

It would be useful to add a dimensioned numbers mechanism to Masque/SQL. This would require both the LQL specification of chapter 3 and the LQL to SQL translation algorithm to be modified, as they currently do not support dimensioned numbers.

type-B predicate declarations: In the current Masque/SQL version, the meaning of each type-B domain-dependent predicate has to be declared in the *interface with data* using an internal pseudo-SQL language (see section 6.2). It shouldn't be too difficult to add a preprocessor, that would allow type-B predicates to be declared using ordinary SQL, and would transform the SQL declarations into the internal pseudo-SQL format.

other SQL datatypes: Masque/SQL currently allows only string and numeric datatypes to be used in the external relational database (see section 6.2). It would be useful to modify the system to support the full range of SQL datatypes (e.g. dates, money etc.). This would probably require adjusting the LQL specification and the translation algorithm, to allow the new datatypes. The type-A declarations in the *interface with data* would also need to be enhanced.

ignorance checking: As it was explained in section 6.2, Masque/SQL is less informative than Masque/PRO, in cases where questions cannot be answered because the database does not contain any suitable data ('ignorance cases'). Masque/PRO keeps internal 'ignorance notes' for any failed `must_satisfy` goal (see the example of section 6.2); these notes help the system to identify the subparts of the query that shouldn't have failed.

Instead, in Masque/SQL an overall SQL query is generated, which is executed as a whole against the relational database. If no rows are retrieved, Masque/SQL can only reply '`No answers found.`', since it doesn't know which particular subparts of the query caused the overall query to fail, and which of these failed subqueries shouldn't have failed.

A possible way to add ignorance checking capabilities to Masque/SQL would be to save the SQL queries that correspond to `must_satisfy` goals[1], during the LQL to SQL translation. Recall that the translation algorithm is recursive in nature: To translate an LQL conjunction list, first each predicate instance of the conjunction list is translated into an SQL `select`, and then these partial `select`s are joined to form the `select` that corresponds to the full conjunction list.

For example, consider the question *'is China larger than France?'* The corresponding logical query is:

```
answer([]):- must_satisfy(has_area(france, _10)),
             must_satisfy(has_area(china, _20)),
             greater_than(_20, _10)
```

The first `has_area` instance is translated into:

```
SELECT *
FROM has_area#2 rel1              [stored SELECT 1]
WHERE rel1.arg1 = 'france'
```

The second `has_area` instance is then translated into:

```
SELECT *
FROM has_area#2 rel2              [stored SELECT 2]
WHERE rel2.arg1 = 'china'
```

Since both `has_area` instances are marked as `must_satisfy`, the system would store the SQL translations of both instances.

The overall logical query is translated into:

```
SELECT *
FROM has_area#2 rel1, has_area#2 rel2
WHERE rel1.arg1 = 'france' AND
      rel2.arg1 = 'china' AND
      rel2.arg2 > rel1.arg2
```

[1]The *logical query simplifier* currently replaces every `must_satisfy`(*predicate*) instance by *predicate*; this simplification could be easily eliminated.

Let us suppose that the system does not know the area of China, i.e. that there is no row corresponding to China in the `has_area#2` table. This would cause the SQL query to fail. However, the system should not report '`no`', since China is indeed larger than France, and the query failed only because the system did not know the area of China.

Before reporting '`no`', the system could execute the SQL queries corresponding to `must_satisfy` goals, to check if the query failed because of ignorance. The system would first execute [stored SELECT 1], which would retrieve the area of France. Then [stored SELECT 2] would be executed, which would reveal that the area of China is not known. Hence, the system would report its ignorance, instead of responding that China is not larger than France.

Implementing such a mechanism for Masque/SQL may actually be more difficult than it seems. The problem is that the translation algorithm does not guarantee that the intermediate SQL `selects` (such as the ones corresponding to `must_satisfy` instances) are well-formed SQL commands. For example, the intermediate SQL `selects` often contain `WHERE` conditions referring to tables not declared in the `FROM` clause (see appendix C).

simplifying the LQL queries: The logical queries generated by Masque (both Masque/PRO and Masque/SQL) often contain logically redundant predicate instances. Consider for example the question *'which county's capital is London?'* in the *geogsql* domain. Masque generates the LQL query:

```
answer([_10]):- country(_10),
                capital(london, _10)
```

Since `capital` is declared in the semantic dictionary as:

```
capital(capital_type, country_type)
```

any value of `_10` that satisfies `capital(london, _10)` will also be a country. Hence, the `country(_10)` instance is redundant. Note, however, that some additional declaration would be needed, to inform the system that all entities of type `country_type` satisfy the predicate `country/1`.

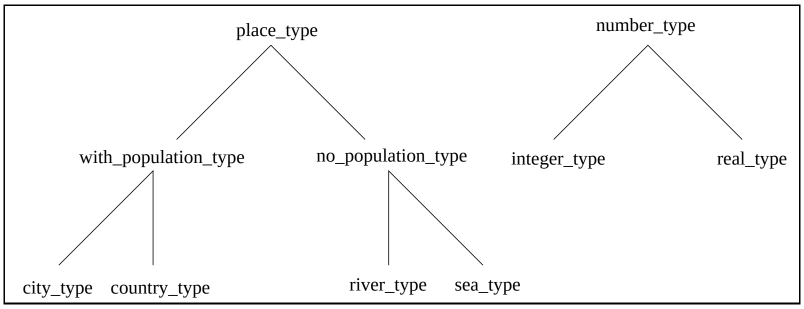

Figure 7–1: A hypothetical type hierarchy

The redundant predicate instances cause the SQL code to be more complicated than necessary. The LQL query of the example would have been translated into:

```
SELECT rel1.arg1
FROM country#1 rel1, capital#2 rel2
WHERE rel2.arg1 = 'london' AND
      rel2.arg2 = rel1.arg1
```

The joining of the two tables, `country#1` and `capital#2`, is redundant. The question could have been answered by executing the SQL query:

```
SELECT rel1.arg1
FROM capital#2 rel1
WHERE rel1.arg1 = 'london'
```

which is the SQL query that would have been generated if the `country` instance had been removed from the LQL query.

The Masque *type hierarchy* can help detect more subtle cases of logically redundant predicate instances. Consider the question *'what is the population of each place'*. The corresponding LQL query is:

```
answer([_10,_20]):- place(_10),
                    population(_20, _10)
```

Assuming that the *type hierarchy* of the domain is the one shown in figure 7–1, and that the *semantic dictionary* contains the declarations:

```
place(place_type)
population(number_type, with_population_type)
```

the `place` instance is redundant, since any value of `_10` that satisfies the `population` instance will be of type `with_population_type` and hence of type `place_type`.

The *logical query simplifier* of Masque/SQL (see figure 6–1) could be extended to handle logically redundant predicate instances, of the kind described above. This would remove redundant joins from the generated SQL code, and since table joins are usually very expensive in terms of execution time, it would also improve the efficiency of the system.

Bibliography

[Androutsopoulos 92] I. Androutsopoulos. Masque/SQL: A Natural Language Front-End for Relational Databases, User's Manual. Department of Artificial Intelligence, University of Edinburgh, 1992.

[Auxerre & Inder 86] P. Auxerre and R. Inder. MASQUE Modular Answering System for Queries in English - User's Manual. Technical Report AIAI/SR/10, Artificial Intelligence Applications Institute, University of Edinburgh, June 1986.

[Auxerre 86] P. Auxerre. MASQUE Modular Answering System for Queries in English - Programmer's Manual. Technical Report AIAI/SR/11, Artificial Intelligence Applications Institute, University of Edinburgh, March 1986.

[Bell & Rowe 92] J.E. Bell and L.A. Rowe. An Exploratory Study of Ad Hoc Query Languages to Databases. In *Proceedings of the 8th International Conference on Data Engineering*, pages 606–613. IEEE Computer Society Press, 1992.

[Ceri *et al* 89] S. Ceri, G. Gottlob, and G. Wiederhold. Efficient Database Access from Prolog. *IEEE Transactions on Software Engineering*, 15(2):153–163, February 1989.

[Ceri *et al* 90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[De Roeck *et al* 90] A.N. De Roeck, H.E. Jowsey, B.G.T. Lowden, R. Turner, and B.R. Walls. A Natural Language Front End to Relational Systems Based on Formal Semantics. In *Proceedings of InfoJapan '90, Tokyo*, 1990.

[De Roeck *et al* 91] A.N. De Roeck, C.J. Fox, B.G.T. Lowden, R. Turner, and B.R. Walls. A Natural Language System Based on Formal Semantics. In *Proceedings of 'Current Issues in Computational Linguistics', Penaug, Malaysia*, 1991.

[Draxler 92] C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates in Logic Programming Languages*. Unpublished PhD thesis, University of Zurich, 1992.

[Fernandes 90] A. Fernandes. A Meaning Representation Language for Natural-Language Interfaces. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1990.

[Fernandes *et al* 91] A. Fernandes, G. Ritchie, and D. Moffat. A Formal Reconstruction of Procedural Semantics. Unpublished paper, Dept. of Artificial Intelligence, University of Edinburgh, August 1991.

[Iliopoulos 92] A. Iliopoulos. Computing Action Plans from Logical Representations in a Natural Language Front-End. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1992.

[ING90a] INGRES Corporation. *INGRES/Embedded SQL User's Guide and Reference Manual*, version 6.3 for UNIX edition, 1990.

[ING90b] INGRES Corporation. *INGRES/SQL Reference Manual*, version 6.3 for UNIX edition, 1990.

[Lindop 86] J.D. Lindop. Enhancements to a Natural Language Front End. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1986.

[Lowden *et al* 91] B.G.T. Lowden, B.R. Walls, A.N. De Roeck, C.J. Fox, and R. Turner. Efficient Generation of SQL Expressions from Natural Language. In *Proceedings of BNCOD '91, Wolverhampton*, 1991.

[Lucas 88] R. Lucas. *Database Applications Using Prolog.* Halsted Press, 1988.

[Minker 88] J. Minker. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufman, 1988.

[Ott 92] N. Ott. Aspects of the Automatic Generation of SQL Statements in a Natural Language Query Interface. *Information Systems*, 17(2):147–159, 1992.

[Pereira & Shieber 87] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis.* Stanford Center for the Study of Language and Information (CSLI), 1987.

[Pereira & Warren 80] F. Pereira and D.H.D. Warren. Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13:231–278, 1980.

[Pereira 81] F. Pereira. Extraposition Grammars. *Computational Linguistics*, 7(4), October-December 1981.

[Pereira 82] F. Pereira. *Logic for Natural Language Analysis.* Unpublished PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1982. Also as SRI International, Technical Note 275, January 1983.

[Sentance 89] S. Sentance. Improved Responses from an English Language Front End. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[Spenceley 89] S. Spenceley. Imperatives and Temporal Question Answering in an English Language Front End. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[Ullman 88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems– Volume 1.* Computer Science Press, 1988.

[Warren & Pereira 82] D. Warren and F. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, July-December 1982.

[Warren 81] D.H.D. Warren. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1981. Also in proceedings of the 7th International Conference on Very Large Databases, France, 1981.

Appendix A

Context-Free Grammar Description of the LQL Syntax

The following context-free grammar partially defines the syntax of LQL. To keep the grammar simple, some of the LQL syntax restrictions discussed in 3.3.9 (e.g. the restriction that a `max` instance must be preceded by a `setof` instance 'binding' the second argument of `max`) are omitted.

- The special symbol of the grammar is *logical_query*.
- 'e' is the empty string.
- Terminal symbols are written in `this typeface`.
- Non-terminal symbols are written in $this\ typeface$.
- Symbols not defined formally are enclosed in $\ll\gg$.
- For simplicity, the following meta-rule is used. For any production rule of the form

  $$non\_terminal \rightarrow \ldots$$

  the production rule

  $$non\_terminal^{*} \rightarrow \ldots |var$$

  is assumed.
- The reader will notice that the CF grammar defines some LQL syntax elements (e.g. LQL pairs or LQL sets) that are not used in the definition of *logical_query*. These syntax elements are the possible values that can be assigned to LQL variables during evaluation. They are included in the CF grammar for completeness.

$$
\begin{array}{rcl}
\mathit{integer} & \rightarrow & \ll \mathit{signed\_integer} \gg \;|\; \ll \mathit{unsigned\_integer} \gg \\
\mathit{real} & \rightarrow & \ll \mathit{signed\_real} \gg \;|\; \ll \mathit{unsigned\_real} \gg \\
\mathit{number} & \rightarrow & \mathit{integer} \mid \mathit{real} \\
\mathit{var} & \rightarrow & \_\,\mathit{unsigned\_integer} \\
\mathit{atom} & \rightarrow & \mathit{number} \mid \; \ll \mathit{symbol} \gg \;|\; \ll \mathit{time} \gg \\
\mathit{pair} & \rightarrow & \mathit{number}^{*} : \mathit{atom}^{*} \\
\mathit{set} & \rightarrow & \mathit{atom\_set} \mid \mathit{pair\_set} \mid [\,] \\
\mathit{atom\_set} & \rightarrow & [\mathit{atom\_list}] \\
\mathit{pair\_set} & \rightarrow & [\mathit{pair\_list}] \\
\mathit{atom\_list} & \rightarrow & \mathit{atom}^{*} \mid \mathit{atom}^{*}, \mathit{atom\_list} \\
\mathit{pair\_list} & \rightarrow & \mathit{pair}^{*} \mid \mathit{pair}^{*}, \mathit{pair\_list} \\
\mathit{functor} & \rightarrow & \ll \mathit{symbol\_with\_no\_upper\_case\_letter} \gg \\
\mathit{predicate\_inst} & \rightarrow & \mathit{domain\_dep\_pred} \mid \mathit{built\_in\_pred} \\
\mathit{domain\_dep\_pred} & \rightarrow & \mathit{type\_A\_pred} \mid \mathit{type\_B\_pred} \\
\mathit{type\_A\_pred} & \rightarrow & \mathit{functor}(\mathit{atom\_list}) \\
\mathit{type\_B\_pred} & \rightarrow & \mathit{functor}(\mathit{atom}^{*}, \mathit{var}) \\
\mathit{built\_in\_pred} & \rightarrow & \texttt{true} \mid \\
 & & \texttt{length}(\mathit{var}, \ll \mathit{unsigned\_integer} \gg^{*}) \mid \\
 & & \texttt{is}(\mathit{var}, \mathit{var}) \mid \\
 & & \texttt{min}(\mathit{atom}^{*}, \mathit{var}) \mid \\
 & & \texttt{max}(\mathit{atom}^{*}, \mathit{var}) \mid \\
 & & \texttt{tot}(\mathit{number}^{*}, \mathit{var}) \mid \\
 & & \texttt{av}(\mathit{number}^{*}, \mathit{var}) \mid \\
 & & \texttt{setof}(\mathit{var}, \mathit{quant\_conj\_list}, \mathit{var}) \mid \\
 & & \texttt{setof}(\mathit{var} : \mathit{var}, \mathit{quant\_conj\_list}, \mathit{var}) \mid \\
 & & \texttt{setof}(\mathit{var} : [\mathit{var}], \mathit{quant\_conj\_list}, \mathit{var}) \mid \\
 & & \texttt{\textbackslash+}(\mathit{conjunction\_list}) \mid \\
 & & \texttt{greater\_than}(\mathit{var}, \mathit{var}) \mid \\
 & & \texttt{less\_than}(\mathit{var}, \mathit{var}) \mid \\
 & & \mathit{var}\texttt{>}\mathit{number} \mid \\
 & & \mathit{var}\texttt{<}\mathit{number} \\
\mathit{conjunction\_list} & \rightarrow & \mathit{predicate\_inst} \mid \\
 & & \mathit{conjunction\_list}, \mathit{conjunction\_list} \mid \\
 & & \mathit{quant\_conj\_list} \\
\mathit{quant\_conj\_list} & \rightarrow & \mathit{q\_list}(\mathit{conjunction\_list}) \\
\mathit{qlist} & \rightarrow & e \mid \mathit{var}\hat{\ }\mathit{qlist} \\
\mathit{logical\_query} & \rightarrow & \texttt{answer}([\,]) :- \mathit{conjunction\_list} \mid \\
 & & \texttt{answer}([\mathit{var}]) :- \mathit{conjunction\_list} \mid \\
 & & \texttt{answer}([\mathit{var}, \mathit{var}]) :- \mathit{conjunction\_list} \mid \\
 & & \texttt{answer}([\mathit{var}\#\mathit{var}]) :- \mathit{conjunction\_list}
\end{array}
$$

Appendix B

LQL to SQL Translation Examples

This appendix provides some step-by-step LQL to SQL translation examples, using the algorithm described in section 5. To make the code more readable, some of the brackets in the SQL code generated by the algorithm are omitted.

'is there some ocean that does not border any country?'

This is an example of a yes/no query. The LQL query produced by Masque is:

```
answer([]):-
        ocean(_11343),
        \+ (country(_11344),
            borders(_11343,_11344))
```

1. The bindings structure BS is initially empty.

2. `ocean(_11343)` is translated into:

   ```
   SELECT *
   FROM ocean-1 REL1
   ```

   and the BS becomes:

BS	
variable	binding
_11343	REL1.ARG1

3. The translation of the \+ instance begins. A copy BS' of BS is kept.

BS'	
variable	binding
_11343	REL1.ARG1

4. `country(_11344)` is translated into:

```
SELECT *
FROM country-1 REL2
```

and BS becomes:

BS	
variable	binding
_11343	REL1.ARG1
_11344	REL2.ARG1

5. `borders(_11343,_11344)` is translated into:

```
SELECT *
FROM borders-2 REL3
WHERE  REL3.ARG1 = REL1.ARG1 AND
       REL3.ARG2 = REL2.ARG2
```

6. The conjunction list:

```
country(_11344),
borders(_11343,_11344)
```

is translated into:

```
SELECT *
FROM country-1 REL2, borders-2 REL3
WHERE  REL3.ARG1 = REL1.ARG1 AND
       REL3.ARG2 = REL2.ARG2
```

7. The quantified conjunction list:

```
(country(_11344),
 borders(_11343,_11344))
```

has the same translation as its body.

8. BS' becomes the new BS:

BS	
variable	binding
_11343	REL1.ARG1

9. The \+ instance is translated into:

```
SELECT
FROM
WHERE NOT EXISTS ( SELECT *
                   FROM country-1 REL2, borders-2 REL3
                   WHERE  REL3.ARG1 = REL1.ARG1 AND
                          REL3.ARG2 = REL2.ARG2 )
```

10. The conjunction list:

```
ocean(_11343),
\+ (country(_11344),
    borders(_11343,_11344))
```

is translated into:

```
SELECT *
FROM ocean-1 REL1
WHERE  NOT EXISTS ( SELECT *
                    FROM country-1 REL2, borders-2 REL3
                    WHERE  REL3.ARG1 = REL1.ARG1 AND
                           REL3.ARG2 = REL2.ARG2 )
```

11. The complete LQL query has the same translation as the conjunction list.

'how large is the smallest american country?'

This is an example of a complex enumeration query. The LQL produced by Masque is:

```
answer([_9100]):-
        (setof(_9103:_9104,
               (country(_9104),
                has_area(_9104,_9103),
                american(_9104)),
               _9102),
         min(_9101,_9102)),
        has_area(_9101,_9100)
```

1. The translation of the `setof` instance begins. A copy of BS (i.e. an empty bindings structure) BS' is kept.

2. `country(_9104)` is translated into:

```
SELECT *
FROM country-1 REL1
```

BS	
variable	binding
_9104	REL1.ARG1

3. `has_area(_9104,_9103)` is translated into:

```
SELECT *
FROM has_area-2 REL2
WHERE REL2.ARG1 = REL1.ARG1
```

BS	
variable	binding
_9104	REL1.ARG1
_9103	REL2.ARG2

4. `american(_9104)` is translated into:

```
SELECT *
FROM american-1 REL3
WHERE REL3.ARG1 = REL1.ARG1
```

5. The quantified conjunction list:

```
(country(_9104),
 has_area(_9104,_9103),
 american(_9104))
```

   is translated into:

```
SELECT *
FROM country-1 REL1, has_area-2 REL2, american-1 REL3
WHERE  REL2.ARG1 = REL1.ARG1 AND
       REL3.ARG1 = REL1.ARG1
```

6. The `setof` instance is translated into the empty string, but as a side-effect:

```
SELECT DISTINCT REL2.ARG2, REL1.ARG1
FROM country-1 REL1, has_area-2 REL2, american-1 REL3
WHERE  REL2.ARG1 = REL1.ARG1 AND
       REL3.ARG1 = REL1.ARG1
```

   is inserted in BS' (which was empty) as the binding of _9102, and BS' becomes the new BS:

BS	
variable	binding

_9102	`SELECT DISTINCT REL2.ARG2, REL1.ARG1`
	`FROM country-1 REL1, has_area-2 REL2, american-1 REL3`
	`WHERE  REL2.ARG1 = REL1.ARG1 AND`
	`       REL3.ARG1 = REL1.ARG1`

7. `min` is defined as:

```
SELECT DISTINCT SECOND
FROM PAIR_SET
WHERE FIRST = ( SELECT min(FIRST)
                FROM PAIR_SET )
```

   Since _9101 has no binding in BS, the instance `min(_9101,_9102)` is translated into the empty string. However, as a side-effect:

```
SELECT DISTINCT REL1.ARG1
FROM country-1 REL1, has_area-2 REL2, american-1 REL3
WHERE  REL2.ARG1 = REL1.ARG1 AND
       REL3.ARG1 = REL1.ARG1 AND
       REL2.ARG2 = ( SELECT min(REL2.ARG2)
                     FROM  country-1 REL1,
                           has_area-2 REL2,
                           american-1 REL3
                     WHERE  REL2.ARG1 = REL1.ARG1 AND
                            REL3.ARG1 = REL1.ARG1 )
```

is inserted into BS as the binding of _9101:

BS	
variable	binding
_9102	SELECT DISTINCT REL2.ARG2, REL1.ARG1 FROM country-1 REL1, has_area-2 REL2, american-1 REL3 WHERE REL2.ARG1 = REL1.ARG1 AND REL3.ARG1 = REL1.ARG1
_9101	SELECT DISTINCT REL1.ARG1 FROM country-1 REL1, has_area-2 REL2, american-1 REL3 WHERE REL2.ARG1 = REL1.ARG1 AND REL3.ARG1 = REL1.ARG1 AND REL2.ARG2 = (SELECT min(REL2.ARG2) FROM country-1 REL1, has_area-2 REL2, american-1 REL3 WHERE REL2.ARG1 = REL1.ARG1 AND REL3.ARG1 = REL1.ARG1)

8. The quantified conjunction list:

```
(setof(_9103:_9104,
       (country(_9104),
        must_satisfy(has_area(_9104,_9103)),
        american(_9104)),
       _9102))
```

is translated into the empty string.

9. `has_area(_9101,_9100)` is translated into:

```
SELECT *
FROM has_area-2 REL4
WHERE REL4.ARG1 IN (
```

```
SELECT DISTINCT REL1.ARG1
FROM country-1 REL1, has_area-2 REL2, american-1 REL3
WHERE  REL2.ARG1 = REL1.ARG1 AND
       REL3.ARG1 = REL1.ARG1 AND
       REL2.ARG2 = ( SELECT min(REL2.ARG2)
                     FROM  country-1 REL1,
                           has_area-2 REL2,
                           american-1 REL3
                     WHERE  REL2.ARG1 = REL1.ARG1 AND
                            REL3.ARG1 = REL1.ARG1 ) )
```

and the bindings structure becomes:

BS	
variable	binding
_9102	`SELECT DISTINCT REL2.ARG2, REL1.ARG1 FROM country-1 REL1, has_area-2 REL2, american-1 REL3 WHERE REL2.ARG1 = REL1.ARG1 AND REL3.ARG1 = REL1.ARG1`
_9101	`SELECT DISTINCT REL1.ARG1 FROM country-1 REL1, has_area-2 REL2, american-1 REL3 WHERE REL2.ARG1 = REL1.ARG1 AND REL3.ARG1 = REL1.ARG1 AND REL2.ARG2 = ( SELECT min(REL2.ARG2) FROM country-1 REL1, has_area-2 REL2, american-1 REL3 WHERE REL2.ARG1 = REL1.ARG1 AND REL3.ARG1 = REL1.ARG1 )`
_9100	REL4.ARG2

10. The conjunction list:

```
(setof(_9103:_9104,
               (country(_9104),
                must_satisfy(has_area(_9104,_9103)),
                american(_9104)),
               _9102),
         min(_9101,_9102)),
        has_area(_9101,_9100)
```

has the same translation as the `has_area` instance.

11. Since *bndg*(_9100) does not have the form:

```
SELECT ...
FROM ...
[WHERE ...]
```

the complete LQL query is translated into:

```
SELECT DISTINCT REL4.ARG2
FROM has_area-2 REL4
WHERE REL4.ARG1 IN (

SELECT DISTINCT REL1.ARG1
FROM country-1 REL1, has_area-2 REL2, american-1 REL3
WHERE  REL2.ARG1 = REL1.ARG1 AND
       REL3.ARG1 = REL1.ARG1 AND
       REL2.ARG2 = ( SELECT min(REL2.ARG2)
                     FROM  country-1 REL1,
                           has_area-2 REL2,
                           american-1 REL3
                     WHERE  REL2.ARG1 = REL1.ARG1 AND
                            REL3.ARG1 = REL1.ARG1 ) )
```

'what is the average area of the countries in each continent?'

This is an example of a complex enumeration query. The LQL query produced by Masque is:

```
answer([_12800,_12801]):-
        continent(_12800),
        setof(_12804:[_12805],
              (area(_12804,_12805),
               country(_12805),
               in(_12805,_12800)),
              _12803),
        av(_12801,_12803)
```

1. The bindings structure BS is initially empty.

2. `continent(_12800)` is translated into:

```
SELECT *
FROM continent-1 REL1
```

BS	
variable	binding
_12800	REL1.ARG1

3. The translation of the `setof` instance begins. A copy BS' of the bindings structure BS is kept:

BS'	
variable	binding
_12800	REL1.ARG1

4. `area(_12804,_12805)` is translated into:

```
SELECT *
FROM area-2 REL2
```

BS	
variable	binding
_12800	REL1.ARG1
_12804	REL2.ARG1
_12805	REL2.ARG2

5. `country(_12805)` is translated into:

```
SELECT *
FROM country-1 REL3
WHERE REL3.ARG1 = REL2.ARG2
```

6. `in(_12805,_12800)` is translated into:

```
SELECT *
FROM in-2 REL4
WHERE  REL4.ARG1 = REL2.ARG2 AND
       REL4.ARG2 = REL1.ARG1
```

7. The conjunction list:

```
(area(_12804,_12805),
 country(_12805),
 in(_12805,_12800))
```

   is translated into:

```
SELECT *
FROM area-2 REL2, country-1 REL3, in-2 REL4
WHERE  REL3.ARG1 = REL2.ARG2 AND
       REL4.ARG1 = REL2.ARG2 AND
       REL4.ARG2 = REL1.ARG1
```

8. The `setof` instance is translated into the empty string, but as a side-effect:

```
SELECT DISTINCT REL2.ARG1, REL2.ARG2
FROM area-2 REL2, country-1 REL3, in-2 REL4
WHERE  REL3.ARG1 = REL2.ARG2 AND
       REL4.ARG1 = REL2.ARG2 AND
       REL4.ARG2 = REL1.ARG1
```

   is inserted into BS' as the binding of _12803:

BS'	
variable	binding
_12800	REL1.ARG1
_12803	SELECT DISTINCT REL2.ARG1, REL2.ARG2 FROM area-2 REL2, country-1 REL3, in-2 REL4 WHERE REL3.ARG1 = REL2.ARG2 AND REL4.ARG1 = REL2.ARG2 AND REL4.ARG2 = REL1.ARG1

and BS' becomes the new BS:

BS	
variable	binding
_12800	REL1.ARG1
_12803	SELECT DISTINCT REL2.ARG1, REL2.ARG2 FROM area-2 REL2, country-1 REL3, in-2 REL4 WHERE REL3.ARG1 = REL2.ARG2 AND REL4.ARG1 = REL2.ARG2 AND REL4.ARG2 = REL1.ARG1

9. `av` is defined as:

```
SELECT avg(FIRST)
FROM PAIR_SET
```

10. Since _12801 has no binding in BS, the instance `av(_12801,_12803)` is translated into the empty string. However, as a side-effect:

```
SELECT avg(REL2.ARG1)
FROM area-2 REL2, country-1 REL3, in-2 REL4
WHERE  REL3.ARG1 = REL2.ARG2 AND
       REL4.ARG1 = REL2.ARG2 AND
       REL4.ARG2 = REL1.ARG1
```

is inserted into the BS as the binding of _12801:

BS	
variable	binding
_12800	REL1.ARG1
_12803	SELECT DISTINCT REL2.ARG1, REL2.ARG2 FROM area#2 REL2, country-1 REL3, in-2 REL4 WHERE REL3.ARG1 = REL2.ARG2 AND REL4.ARG1 = REL2.ARG2 AND REL4.ARG2 = REL1.ARG1
_12801	SELECT avg(REL2.ARG1) FROM area-2 REL2, country-1 REL3, in-2 REL4 WHERE REL3.ARG1 = REL2.ARG2 AND REL4.ARG1 = REL2.ARG2 AND REL4.ARG2 = REL1.ARG1

11. The conjunction list:

```
continent(_12800),
setof(_12804:[_12805],
      (area(_12804,_12805),
       country(_12805),
       in(_12805,_12800)),
      _12803),
av(_12801,_12803)
```

has the same translation as `continent(_12800)`:

```
SELECT *
FROM continent-1 REL1
```

12. Since *bndg*(_12801) is a `SELECT` with an aggregate (`avg`), the LQL query is translated into:

```
SELECT DISTINCT REL1.ARG1, avg(REL2.ARG1)
FROM continent-1 REL1, area-2 REL2, country-1 REL3, in-2 REL4
WHERE  REL3.ARG1 = REL2.ARG2 AND
       REL4.ARG1 = REL2.ARG2 AND
       REL4.ARG2 = REL1.ARG1
GROUP BY REL1.ARG1
```

Appendix C

Masque/SQL Questions and Answers

This appendix shows some sample user questions and the corresponding responses generated by Masque/SQL. All questions refer to the *geogsql* world geography domain[1] (see chapter 6). For each question, the corresponding LQL and SQL queries are also given. Masque/SQL reports the LQL queries in a pretty-printed format, which is slightly different from the internal format described in chapter 3. The mapping between the two formats should be obvious. All times are CPU times. User entries are printed in *this typeface*. Masque/SQL messages are printed in `this typeface`.

`QAS>` *what is the capital of Greece?*

```
LQL Query :
answer([B]) :-
   capital(B,greece)

SQL query :
SELECT DISTINCT rel1.arg1
FROM capital#2 rel1
WHERE rel1.arg2 = 'greece'

athens

1 solution(s)

  Total time used: 1.959 sec.
```

[1] *geogsql* is based on geographical data that were valid in the late 70s.

QAS> *which country's capital is London?*

```
LQL Query :
answer([B]) :-
   country(B)
 & capital(london,B)

SQL query :
SELECT DISTINCT rel1.arg1
FROM country#1 rel1, capital#2 rel2
WHERE rel2.arg1 = ‘london‘ AND
      rel2.arg2 = rel1.arg1

united_kingdom

1 solution(s)

  Total time used: 1.33 sec.
```

QAS> *is China larger than France?*

```
LQL Query :
answer([]) :-
   has_area(france,B)
 & has_area(china,C)
 & greater_than(C,B)

SQL query :
SELECT *
FROM has_area#2 rel1, has_area#2 rel2
WHERE rel1.arg1 = ‘france‘ AND
      rel2.arg1 = ‘china‘ AND
      rel2.arg2 > rel1.arg2


Yes

  Total time used: 0.55 sec.
```

QAS> *is the population of China greater than the population of France?*

```
LQL Query :
answer([]) :-
   population(B,france)
 & population(C,china)
 & greater_than(C,B)
 & true

SQL query :
SELECT *
```

```
FROM population#2 rel1, population#2 rel2
WHERE rel1.arg2 = ‘france‘ AND
      rel2.arg2 = ‘china‘ AND
      rel2.arg1 > rel1.arg1


Yes

  Total time used: 3.19 sec.
```

QAS> *is there some ocean that does not border any country?*

```
LQL Query :
answer([]) :-
   ocean(B)
 & \+
     country(C)
   & borders(B,C)

SQL query :
SELECT *
FROM ocean#1 rel1
WHERE NOT EXISTS (SELECT *
                  FROM country#1 rel2, borders#2 rel3
                  WHERE rel3.arg1 = rel1.arg1 AND
                        rel3.arg2 = rel2.arg1 )
```

Yes, do you want to know more (y/n) ? *yes*

```
LQL Query :
answer([B]) :-
   B = setof C
     ocean(C)
   & \+
       country(D)
     & borders(C,D)

SQL query :
SELECT DISTINCT rel1.arg1
FROM ocean#1 rel1
WHERE NOT EXISTS (SELECT *
                  FROM country#1 rel2, borders#2 rel3
                  WHERE rel3.arg1 = rel1.arg1 AND
                        rel3.arg2 = rel2.arg1 )

southern_ocean

1 solution(s)

  Total time used: 2.66 sec.
```

QAS> *what is the largest country in Europe?*

```
LQL Query :
answer([B]) :-
   E = setof C:D
     country(D)
   & has_area(D,C)
   & in(D,europe)
 & max(B,E)

SQL query :
SELECT DISTINCT rel1.arg1
FROM country#1 rel1, has_area#2 rel2, in#2 rel3
WHERE (rel2.arg1 = rel1.arg1 AND
       rel3.arg1 = rel1.arg1 AND
       rel3.arg2 = ‘europe‘ ) AND
      (rel2.arg2 = (SELECT max(rel2.arg2)
                    FROM country#1 rel1, has_area#2 rel2, in#2 rel3
                    WHERE rel2.arg1 = rel1.arg1 AND
                          rel3.arg1 = rel1.arg1 AND
                          rel3.arg2 = ‘europe‘ ) )

france

1 solution(s)

  Total time used: 2.8 sec.
```

QAS> *how large is the smallest american country?*

```
LQL Query :
answer([B]) :-
   E = setof C:D
     country(D)
   & has_area(D,C)
   & american(D)
 & min(F,E)
 & has_area(F,B)

SQL query :
SELECT DISTINCT rel4.arg2
FROM has_area#2 rel4
WHERE rel4.arg1 IN (SELECT rel1.arg1
                    FROM country#1 rel1, has_area#2 rel2, american#1 rel3
                    WHERE (rel2.arg1 = rel1.arg1 AND
                           rel3.arg1 = rel1.arg1 ) AND
                          (rel2.arg2 = (SELECT min(rel2.arg2)
                                        FROM country#1 rel1,
     has_area#2 rel2,
     american#1 rel3
                                        WHERE rel2.arg1 = rel1.arg1 AND
```

```
                                            rel3.arg1 = rel1.arg1 ) ) )

133

1 solution(s)

  Total time used: 1.39 sec.
```

QAS> *what is the ocean that borders african countries and that borders asian countries?*

```
LQL Query :
answer([B]) :-
   ocean(B)
 & country(C)
 & african(C)
 & borders(B,C)
 & country(D)
 & asian(D)
 & borders(B,D)

SQL query :
SELECT DISTINCT rel1.arg1
FROM ocean#1 rel1, country#1 rel2,
     african#1 rel3, borders#2 rel4,
     country#1 rel5, asian#1 rel6,
     borders#2 rel7
WHERE rel3.arg1 = rel2.arg1 AND
      rel4.arg1 = rel1.arg1 AND
      rel4.arg2 = rel2.arg1 AND
      rel6.arg1 = rel5.arg1 AND
      rel7.arg1 = rel1.arg1 AND
      rel7.arg2 = rel5.arg1

indian_ocean

1 solution(s)

  Total time used: 4.88 sec.
```

QAS> *what are the capitals of the countries bordering the Baltic?*

```
LQL Query :
answer([B]) :-
   B = setof C
     exists D
       D = setof E
         country(E)
       & borders(E,baltic)
     & capital(C,D)
```

```
SQL query :
SELECT DISTINCT rel3.arg1
FROM capital#2 rel3
WHERE rel3.arg2 IN (SELECT rel1.arg1
                    FROM country#1 rel1, borders#2 rel2
                    WHERE rel2.arg1 = rel1.arg1 AND
                          rel2.arg2 = ‘baltic‘ )

bonn
copenhagen
east_berlin
helsinki
moscow
stockholm
warsaw

7 solution(s)

  Total time used: 4.65 sec.
```

QAS> *what is the total area of the countries in Europe?*

```
LQL Query :
answer([B]) :-
   E = setof C:[D]
     area(C,D)
   & country(D)
   & in(D,europe)
 & tot(B,E)

SQL query :
SELECT DISTINCT sum(rel1.arg1)
FROM area#2 rel1, country#1 rel2, in#2 rel3
WHERE rel2.arg1 = rel1.arg2 AND
      rel3.arg1 = rel1.arg2 AND
      rel3.arg2 = ‘europe‘

1881886

1 solution(s)

  Total time used: 4.45 sec.
```

QAS> *what is the average area of the countries in each continent?*

```
LQL Query :
answer([B,C]) :-
   continent(B)
 & F = setof D:[E]
     area(D,E)
   & country(E)
```

```
   & in(E,B)
 & av(C,F)

SQL query :
SELECT DISTINCT rel1.arg1, avg(rel2.arg1)
FROM continent#1 rel1, area#2 rel2, country#1 rel3, in#2 rel4
WHERE rel3.arg1 = rel2.arg2 AND
      rel4.arg1 = rel2.arg2 AND
      rel4.arg2 = rel1.arg1
GROUP BY rel1.arg1

africa 234090
america 496712
asia 485621
australasia 543940
europe 58808.9

5 solution(s)

  Total time used: 5.35 sec.
```

QAS> *what is the population of the largest country in each continent?*

```
LQL Query :
answer([B,C]) :-
   continent(B)
 & F = setof D:E
     country(E)
   & has_area(E,D)
   & in(E,B)
 & max(G,F)
 & population(C,G)

SQL query :
SELECT DISTINCT rel1.arg1, rel5.arg1
FROM continent#1 rel1, population#2 rel5
WHERE rel5.arg2 IN (SELECT rel2.arg1
                    FROM country#1 rel2, has_area#2 rel3, in#2 rel4
                    WHERE (rel3.arg1 = rel2.arg1 AND
                           rel4.arg1 = rel2.arg1 AND
                           rel4.arg2 = rel1.arg1 ) AND
                          (rel3.arg2 = (SELECT max(rel3.arg2)
                                        FROM country#1 rel2,
                                             has_area#2 rel3,
                                             in#2 rel4
                                        WHERE rel3.arg1 = rel2.arg1 AND
                                              rel4.arg1 = rel2.arg1 AND
                                              rel4.arg2 = rel1.arg1 ) ) )

africa 16900000
america 22047000
asia 250900000
```

```
australasia 13268000
europe 52350000

5 solution(s)

  Total time used: 6.05 sec.
```

QAS> *what is the population and the capital of the largest country in each continent?*

```
I do not understand the structure of this sentence.

  Total time used: 0.44 sec.
```

QAS> *what are the countries from which a river flows into the Black Sea?*

```
LQL Query :
answer([B]) :-
   B = setof C
     exists D
       country(C)
     & river(D)
     & flows(D,C,black_sea)

SQL query :
SELECT DISTINCT rel1.arg1
FROM country#1 rel1, river#1 rel2, flows#3 rel3
WHERE rel3.arg1 = rel2.arg1 AND
      rel3.arg2 = rel1.arg1 AND
      rel3.arg3 = 'black_sea'

austria
czechoslovakia
hungary
romania
soviet_union
west_germany
yugoslavia

7 solution(s)

  Total time used: 2.4700000000000002 sec.
```

QAS> *which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India?*

```
LQL Query :
answer([B]) :-
   country(B)
 & borders(B,mediterranean)
 & country(C)
```

```
 & country(D)
 & population(E,D)
 & population(F,india)
 & greater_than(E,F)
 & borders(D,C)
 & borders(B,C)

SQL query :
SELECT DISTINCT rel1.arg1
FROM country#1 rel1, borders#2 rel2, country#1 rel3,
     country#1 rel4, population#2 rel5, population#2 rel6,
     borders#2 rel7, borders#2 rel8
WHERE rel2.arg1 = rel1.arg1 AND
      rel2.arg2 = ‘mediterranean‘ AND
      rel5.arg2 = rel4.arg1 AND
      rel6.arg2 = ‘india‘ AND
      rel5.arg1 > rel6.arg1 AND
      rel7.arg1 = rel4.arg1 AND
      rel7.arg2 = rel3.arg1 AND
      rel8.arg1 = rel1.arg1 AND
      rel8.arg2 = rel3.arg1

turkey

1 solution(s)

  Total time used: 23.43 sec.
```

QAS> *what is the largest continent?*

```
LQL Query :
answer([B]) :-
   E = setof C:D
     continent(D)
   & has_area(D,C)
 & max(B,E)

SQL query :
SELECT DISTINCT rel1.arg1
FROM continent#1 rel1, has_area#2 rel2
WHERE (rel2.arg1 = rel1.arg1 ) AND
      (rel2.arg2 = (SELECT max(rel2.arg2)
                    FROM continent#1 rel1, has_area#2 rel2
                    WHERE rel2.arg1 = rel1.arg1 ) )

No answers found.

  Total time used: 0.841 sec.
```

(Masque/PRO would have responded:

`I do not know the value of has_area applied to any continent.`)

QAS> *is Greece the largest country?*

```
LQL Query :
answer([]) :-
   D = setof B:C
     country(C)
   & has_area(C,B)
 & max(greece,D)

SQL query :
SELECT *
FROM dummy_table
WHERE ‘greece‘ IN (SELECT rel1.arg1
                   FROM country#1 rel1, has_area#2 rel2
                   WHERE (rel2.arg1 = rel1.arg1 ) AND
                         (rel2.arg2 = (SELECT max(rel2.arg2)
                                       FROM country#1 rel1, has_area#2 rel2
                                       WHERE rel2.arg1 = rel1.arg1 ) ) )
```

No, do you want to know what is (y/n) ? *yes*

```
LQL Query :
answer([B]) :-
   E = setof C:D
     country(D)
   & has_area(D,C)
 & max(B,E)

SQL query :
SELECT DISTINCT rel1.arg1
FROM country#1 rel1, has_area#2 rel2
WHERE (rel2.arg1 = rel1.arg1 ) AND
      (rel2.arg2 = (SELECT max(rel2.arg2)
                    FROM country#1 rel1, has_area#2 rel2
                    WHERE rel2.arg1 = rel1.arg1 ) )

soviet_union

1 solution(s)

  Total time used: 1.1 sec.
```

QAS> *how many countries does the Danube flow through?*

```
LQL Query :
answer([B#C]) :-
   C = setof D
     country(D)
   & flows(danube,D)
 & length(C,B)
```

```
SQL query :
SELECT count(*)
FROM country#1 rel1, flows#2 rel2
WHERE rel2.arg1 = ‘danube‘ AND
      rel2.arg2 = rel1.arg1

6

1 solution(s)

  Total time used: 0.78 sec.
```

QAS> *how many oceans are larger than France?*

```
LQL Query :
answer([B#C]) :-
   C = setof D
     exists E F
       ocean(D)
     & has_area(france,F)
     & has_area(D,E)
     & greater_than(E,F)
 & length(C,B)

SQL query :
SELECT count(*)
FROM ocean#1 rel1, has_area#2 rel2, has_area#2 rel3
WHERE rel2.arg1 = ‘france‘ AND
      rel3.arg1 = rel1.arg1 AND
      rel3.arg2 > rel2.arg2

0

1 solution(s)

  Total time used: 1.16 sec.
```

(Masque/PRO would have responded:

`I do not know the value of has_area applied to Southern`

`Ocean, Pacific, Indian Ocean, Atlantic or Arctic Ocean.`)

QAS> *how many countries bordering Hungary are smaller than the largest american country?*

```
LQL Query :
answer([B#C]) :-
   C = setof D
```

```
     exists E F G H
       country(D)
     & borders(D,hungary)
     & G = setof I:J
         country(J)
       & has_area(J,I)
       & american(J)
     & max(H,G)
     & has_area(H,F)
     & has_area(D,E)
     & less_than(E,F)
 & length(C,B)

SQL query :
SELECT count(*)
FROM country#1 rel1, borders#2 rel2, has_area#2 rel6, has_area#2 rel7
WHERE rel2.arg1 = rel1.arg1 AND
      rel2.arg2 = ‘hungary‘ AND
      rel6.arg1 IN (SELECT rel3.arg1
                    FROM country#1 rel3, has_area#2 rel4, american#1 rel5
                    WHERE (rel4.arg1 = rel3.arg1 AND
                           rel5.arg1 = rel3.arg1 ) AND
                          (rel4.arg2 = (SELECT max(rel4.arg2)
                                        FROM country#1 rel3,
                                             has_area#2 rel4,
                                             american#1 rel5
                                        WHERE rel4.arg1 = rel3.arg1 AND
                                              rel5.arg1 = rel3.arg1 ) ) ) AND
      rel7.arg1 = rel1.arg1 AND
      rel7.arg2 < rel6.arg2

4

1 solution(s)

  Total time used: 3.39 sec.
```

Appendix D

Code listings

D.1 New modules in Masque/SQL

The code listings of this section correspond to the *new* modules added to Masque during this project (see figure 6–1). The files listed are:

- '`lql_to_sql.pl`': Prolog code implementing the *LQL to SQL translator.*
- '`prolog_to_lql.pl`': Prolog code implementing the *logical query simplifier.*
- '`db_server.sc`': C code with embedded SQL, implementing the *Masque database server.*
- '`dbsrv_utils.pl`': Prolog code implementing the *Masque to server tranceiver.*

D.2 The 'geogsql' domain

- '`lexical_dictionary`': The *lexical dictionary* of the *geogsql* domain.
- '`semantic_dictionary`': The *semantic dictionary* of the *geogsql* domain.
- '`hierarchy`': The *type hierarchy* of the *geogsql* domain.
- '`interface_with_data`': The *interface with data* of the *geogsql domain.*
- '`create_tables.sql`': SQL script to create the Ingres tables used in the *geogsql* domain. Serves as a description of the Ingres tables used.