On Compiling Logic Programs into Relational Algebra

by

Saeed M.H. Al-Amoudi

A thesis submitted to the
Faculty of Science
University of Glasgow
for the degree of
Master of Science

Department of Computing Science
University of Glasgow
March 1990

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Dedicated to my parents

Declaration

This thesis is entirely on my original work and no part is done in collaboration. Where the work of others is used, explicit reference is made in the text. No part of this thesis has been, or is being submitted for a degree at any other university.

CONTENTS

Acknowledgements

There are many people that I would like to thank and convey my gratefulness. Foremost are my supervisors, Dr. David J. Harper and Dr. Muffy Thomas, whom provided me, over the years, with the idea and supply of the relevant material to this work. I am grateful to Dr. Simon B. Jones for his advise and comments in the work. I also like to thank my colleagues Roslan, Tengku Mohd, Riaz, Djamal, Khaled, Abdellatif, Hakim, Francis, and Daniel for unforgettable moments that we all shared together as students in this department.

Last but not the least, I would like to thank my wife Lolo, my son Sheak, and my daughter Ammel for their patience and support over the years we spent in Glasgow.

Financial support and study leave were provided by King AbdulAziz University which are highly appreciated and gratefully acknowledged.

ABSTRACT

The combination of logic programming methods and database systems technology will result in knowledge bases of increased size and improved efficiency: this topic has received a lot of attention [Zaniolo 1985, Reiter 1978, Chang 1986, Minker 1978, Henschen 1984, Parker 1986, Brodie 1986]. Our approach to integrating logic programming languages (e.g. PROLOG) and database systems is to compile logic programming languages into conventional relational algebra.

There are many technical problems which must be addressed and solved when compiling logic programs into relational algebra. Mainly, we are interested in the following problems: the finiteness (i.e. safety) of a logic program's executions and the differences between logic programing languages and database systems in data representation and typing systems.

Our approach to safety checking integrates the rule/goal graph of [Ullman 1985] with the magic basis of a variable [Zaniolo 1986]. This approach allows us, effectively, to check the safety of a logic program at compile time, for those programs which are strongly safe. Otherwise, the safety of the program with respect to a query must be checked at execution time.

Relational database systems are well typed, whilst logic programming languages are not. We overcome this difference by adding types to PROLOG (i.e TPROLOG). TPROLOG allows the user to define enumerated types, sub–types, structured types, and variant types.

Our approach to compiling typed logic programs into conventional

relational algebra expressions is to translate the logic program containing complex clauses into an equivalent complex–free program, and then to translate it into a form suitable for storage and manipulation by conventional relational database systems. The normalization of logic programs is achieved by removing complex arguments from facts and rules and replacing them with simplified (i.e. normalized) facts and rules. The normalized facts are stored in a conventional relational database (i.e. extensional database), and the normalized rules are stored in a rule base (i.e. intentional database). The translation of a complex–free program into conventional relational algebra is based on [Reiter 1978, Chang 1986, Henschen 1984, Bancilhon 1986].

Chapter 1: Introduction

§ 1–1 The Problem Definition

Logic programming languages enable us to implement knowledge base systems by virtue of their ability to represent and reason with facts and rules [Gallaire 1984]. Database systems provide the technology for storing, managing, and processing very large collections of data efficiently [Ullman 1981, Date 1981]. However, logic programming languages are simpler to use for expressing queries than database system query languages (e.g. relational algebra expressions) [Gallaire 1984, Brodie 1986, Parker 1986]. It would seem that the combination of logic programming methods and database systems technology will result in knowledge bases of increased size and improved efficiency. Our approach to integrating logic programming languages (e.g. PROLOG) and database systems is to compile logic programming languages into conventional relational algebra.

In order to integrate logic programs and database systems, there are many technical problems which must be solved. The fundamental differences between logic programming languages and database systems, which may cause these problems, are discussed in [Brodie 1986, Parker 1986]. Mainly, we are interested in the following problems:

1 -The finiteness (i.e. safety) of a logic program's execution.

2- The representation of data in logic programming languages and database systems.

3– The weakness of typing systems in logic programming languages.

There are two approaches to tackling the safety problem: we can test the safety of a query with respect to the database (facts and rules) at its time of execution, or we can determine at compile time whether a set of rules guarantees the safety of all queries. In general, both approaches are necessary. Most of the existing approaches tackle the problem at execution time [Ullman 1985, Zaniolo 1986, Tsur 1986].

Krishnamurthy [Krishnamurthy 1988] tackles the compile time problem by introducing the notion of strongly safe datalog programs. A program is strongly safe, if any query to the program is determined (i.e. safe).

Our approach integrates the rule/goal graph of [Ullman 1985] with the magic basis of a variable [Zaniolo 1986]. This approach allows us to effectively check the safety of a logic program at compile time, for those programs which are strongly safe. Otherwise, the safety of a program with respect to a query must be checked at execution time. For example, the PROLOG program in figure 1–2 is strongly safe because of the following:

1) Any query to the facts emp and person is safe.

2) In the rule whose rule head is glaswegian_infant, the body predicates are strongly safe. Therefore, any query to the rule is safe. Similarly, any query to the rule whose rule head is glaswegian_emp is safe.

However, the PROLOG program in figure 1-1, which is used to calculate the factorial of number n, is not strongly safe. For example, the query ?- factorial(-1, N). is unsafe.

```
factorial(0,1).
factorial(N,F) :- N1 is N -1, factorial(N1,F1), F is N*F1.
```

Figure 1–1. A PROLOG program to calculate the factorial.

```
person ( joe, cool,address( none, glasgow), 20).
person (max, fax,address(flat(2l, 18, windsor_street, g20),glasgow), 40).
person (joe, doe, address(house(31, kew_drive, g12), glasgow),3).
emp ( joe, cool, porter, none).
emp (max, fax, guard, [degree1(hs, 1968)]).
emp (fred, red, staff,
    [degree1(hs, 1975), degree2(msc, ba, school(glasgow_university, glasgow), 1980),
    degree2(phd, ba, school(glasgow_university, glasgow), 1983)]).

glaswegian_infant (LN, FN, Age) :-person(FN, LN,address(_, glasgow),Age),
                                    Age < 4.
glaswegian_emp( Ln, Fn,  Sch,Yr) :- emp(Fn, Ln, _, [_,_, degree2(_, _, Sch, Yr)]),
                                     person(Fn, Ln,address(_, glasgow),_),
                                    Yr > 1960,
                                    Yr < 1990.
```

Figure 1–2. A PROLOG program

There exist at least two approaches for compiling logic programs into relational database systems. The first approach assumes that logic programs contain flat (i.e. simple) clauses only [Minker 1978, Jarke 1984, Reiter 1978, Chang 1986]. The other tackles the more general problem of compiling non-flat clauses into algebraic operations on a relational database management system [Zaniolo 1985]. Zaniolo has done this by introducing new relational algebraic operations called Extended Relational

Algebra operations (ERA), and allowing the database systems to store complex facts (i.e. the arguments of facts need not be simple, but they may be complex terms or lists). However, conventional relational database systems can be used to store simple facts only. For example, the fact factorial(0,1) from figure 1–1 is a simple fact. Therefore, we are able to store it in a relational database. The following PROLOG program:

```
person ( joe, cool,address( none, glasgow), 20).
person (max, fax,address(flat(2l, 18, windsor_street, g20),glasgow), 40).
person (joe, doe, address(home(31, kew_drive, g12), glasgow),3).
emp (max, fax, guard, [degree1(hs, 1968)]).
emp (fred, red, staff,
    [degree1(hs, 1975), degree2(msc, ba, school(glasgow_university, glasgow), 1980),
    degree2(phd, ba, school(glasgow_university, glasgow), 1983)]).
```

contains complex facts. Therefore, we cannot store them in a relational database. In order to overcome this problem our approach is proposed.

Our approach compiles logic programs containing non-flat clauses into equivalent flat logic programs, and those can be translated into a form suitable for manipulation by conventional relational database systems. This is achieved by removing complex arguments from facts and rules and replacing them with simplified (i.e. normalized) facts and rules. The normalized facts are stored in a conventional relational database (i.e. extensional database), and the normalized rules are stored in a rule base (i.e. intentional database).

In logic programs, queries are answered by using the built-in theorem prover of the logic programming system. This is achieved without regard to the type of the data. For example, the PROLOG program in figure 1–2 is an untyped program. On the other hand, database query languages are

typed. We will remove this difference by extending PROLOG to become a typed language which we will call TPROLOG.

§1-2 System Overview

Our system is outlined in figure 1–3. In this section we briefly describe the major elements of the system. The detailed description of the system is in chapter 6 and chapter 7.

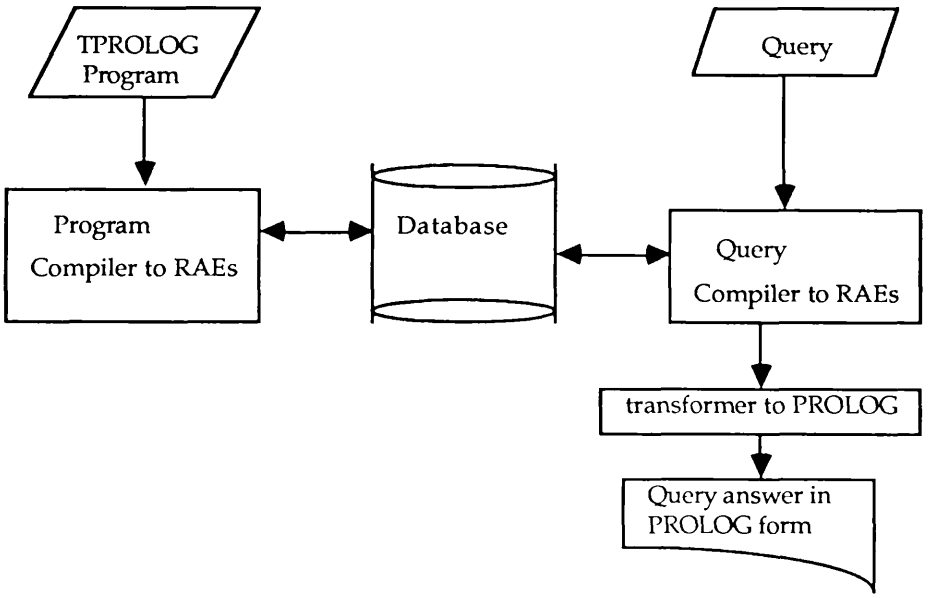


Figure 1–3. The System Overview

Although there are a number of different input types permitted in the system, only two types are described in figure 1–3: TPROLOG program and query.

Briefly, TPROLOG is PROLOG plus a type system. For example, the PROLOG program, in figure 1–2, needs to be extended to include some type information to become a TPROLOG program. For further details refer to Appendix A and chapter 3. Facts of the TPROLOG program are normalized and compiled into relations, and then stored in the extensional database (EDB). Rules in the TPROLOG program are normalized and compiled into a view [Date 1981], and then stored in the intentional database (IDB). Moreover, the IDB, the EDB, and type information are in the database(DB).

Before a TPROLOG program is normalized and compiled into relational algebra and then stored in the DB, it is tested. Every formula (i.e. fact or rule) must be tested to determine if it is well–formed with respect to safety and the type system. If the formula is well–formed, then it is compiled and stored in the DB. Otherwise, it will not be compiled. More specifically, facts are compiled after they are, simplified, type checked and normalized (q.v. chapter 3 and chapter 5). Moreover, types are deduced for rules and are safety–checked (q.v. chapter 3 and chapter 4), before they are finally normalized and compiled into a view (q.v. chapter 5).

A query to the system is written in PROLOG. It is processed by the system. The query process is explained in chapter 6 and chapter 7. A well–formed query is compiled into a set of relational algebraic expressions. A result of such a query is represented in a PROLOG query answering form.

Chapter 2: Logic Programming Languages and Relational Database Systems

§ 2–1 Introduction

Database systems (DBSs) are concerned with how data should be stored and retrieved from a DB. Logic programming Languages (LPLs) (e.g. PROLOG) are concerned with how the data should be represented in a natural way.

The typing systems, data representations, and the execution strategies in LPLs and relational database systems (RDBSs), which we are interested in, are addressed in § 1–1. In this chapter, we discuss problems with these issues in LPLs and RDBSs. Moreover, we discuss with the need to extend LPLs in order to integrate RDBSs and LPLs.

§ 2–2 Logic Programming Languages

§ 2–2–1 Type Checking

A programming language is statically typed, if all type errors of a program can be detected at compile time (e.g. Pascal). In LPLs there is a single domain (i.e. one sort logic) for each program. The domain is defined as a Herbrand universe [Lloyd 1984, Mycroft 1984]. A Herbrand universe may be an infinite domain. Anomalous formulae (i.e. type errors) can be formed in a logic program which have no basis in the real world. LPLs do not provide static type checking. The absence of static type checking makes LPLs unable to provide clean semantics for updating a collection of facts

while preserving integrity [Brodie 1986, Parker 1986].

A programming language is strongly typed, if it prevents a query from being applied to value of an inappropriate type at run time. Queries in logic programming languages are answered by using the built–in theorem prover of the logic programming system [Kowalski 1979] without regard to the type of the data. Hence, LPLs are not strongly typed.

§ 2–2–2 Data Storage and Representation

A logic program is defined by a finite set of first order formulae [Lloyd 1984]. A first order formula, simply a formula, contains a set of terms. A term is either a complex or simple [Zaniolo 1985]. Moreover a formula is either a fact, a rule, or a query [Brodie 1986]. Although LPLs are concerned with how data should be represented in a natural way, all formulae in a logic program are represented independently of each other.

Finally, since answering systems in LPLs give a tuple at a time, the interaction in LPLs with the secondary storage is inefficient [Nussbaum 1989]. Moreover, LPLs do not accommodate multiple users [Parker 1986].

§ 2–2–3 Execution Strategy

A top–down proof method works by resolving the denial of the goal with the original assertions in order to produce an empty clause and thereby prove the goal by refutation. Resolution is a top–down method which uses the modus tollens inference rule [Kowalski 1979]. From the modus tollens inference rule we observe that a top–down method can be carried out even when the original assertions have not yet been asserted.

SL–resolution (linear resolution with selection computation rule [Lloyd 1984]) is a resolution by refutation method. We assume the computation rule is selection, from top to bottom (i.e. depth–first), of the left most atom. One of the deficiencies of SL–resolution that it is not guaranteed to terminate in the sense that it does not guarantee is that an empty clause can be generated for a successful resolution or a non_empty clause can be generated which is not unified with any formula [Kowalski 1975].

§ 2–2–4 PROLOG

PROLOG is a declarative programming language based on first–order Horn clauses [Frost 1986, Lloyd 1984, Kowalski 1979]. A PROLOG program consists of a set of facts and rules [Clocksin 1984].

A fact in a PROLOG program is a ground atom. In general, PROLOG allows a unit clause (i.e. a rule with no body predicates) to be defined [Lloyd 1984] (e.g. person(X,Y):-., where X and Y are variables). A unit clause allows the definition of infinite relations. From a database point of view, facts in PROLOG may be considered as a set of relations in a relational database, in spite of the fact that PROLOG allows infinite relations to be defined. Thus, we may consider the subset of a PROLOG program, which does not contain unit clauses, as a set of relations in a relational database.

PROLOG program rules may be thought of as expressions in a quasi–tuple relational calculus. Each rule is written in the form $q(t) :- \Psi(t).$, which denotes that q contains the set of tuples t that satisfy the predicate Ψ. Note, that the tuples, t, are not necessarily flat.

Queries in PROLOG are answered by resolving them by refutation. Thus, the strategy in PROLOG for computing the answer to a query is a SL–resolution method.

§ 2–3 Relational Database Systems

§ 2–3–1 General

The relations in a relational database are finite relations (e.g. PRTV, and INGRES [Todd 1976, Date 1981, Ullman 1981]). Relational database query languages (e.g. ISBL, ASTRJD, SQL, Query–By–Example, and QUEL [Todd 1976, Bell 1978, Date 1981, Ullman 1981]) are based on either relational algebra, tuple relational calculus, or domain relational calculus.

Those relational database query languages (e.g. ISBL, ASTRJD, and SQL) which are based on relational algebra are procedural languages. They allow end–users to manipulate relations using relational algebraic operators (i.e. join, select, project, etc.) in order to obtain the result they require. The operators used in Relational Algebraic Expressions (RAEs) are applied only to finite relations and the operands are either constants or variables denoting relations of fixed arity.

Those relational database query languages (for example QUEL) which are based on tuple relational calculus are declarative languages. They allow end–users to specify exactly the properties they require without having to specify how the data is to be obtained from the relations available in the database. The tuple relational calculus is based on first–order predicate logic, and expressions in tuple relational calculus are of the form $\{t \mid \Psi(t)\}$ which denotes the set of tuples t that satisfy the predicate Ψ.

Those relational database query languages (e.g. Query–By–Example) which are based on domain relational calculus are built from the same operators as the tuple relational calculus.

It is important to note that expressions in tuple relational calculus may be used to define an infinite relation such as $\{ t \mid \neg R(t) \}$, which denotes all possible tuples that are not in the relation R.

The following theorems from [Ullman 1981] relate tuple relational calculus expressions, relational algebra expressions and domain relational calculus expressions:

Theorem 1: If E is a relational algebra expression, then there is a safe expression in tuple relational calculus equivalent to E.

Theorem 2 : For every safe tuple relational calculus expression there is an equivalent safe domain relational calculus expression.

Theorem 3 : For every safe domain relational calculus expression, there is an equivalent relational algebraic expression.

From the above theorems we conclude that: for every safe tuple relational calculus expression there is an equivalent relational algebraic expression.

§ 2–3–2 Type Checking

A DB definition (i.e. scheme) is used to represent the type of an entity set in a relation [Ullman 1981]. Any processing of an entity set should be

done with respect to its scheme, while preserving integrity [Date 1981, Ullman 1981, Gray 1984]. Therefore, relational database systems are statically typed systems.

Conventional relational database systems do not allow the use of type constructors to define attribute types. Moreover, an entity set is represented by a relation whose relation schema consists of all the attributes of the entity set. For each attribute there is a finite domain which is defined by the data type of the attribute. Therefore, the cartesian product of a relation's attribute types defines the schema of the relation. Thus, RDBSs use many–sorted logic.

Since relations in RDBSs are based on many sorted logic [Gallaire 1984], relational query languages are based on many sorted logic too. A language based on many sorted logic offers a more precise definition of the programs well as imposing some restrictions. These restrictions prevent a query from being applied to value of an inappropriate type. Therefore, relational database query languages are strongly typed.

§ 2–3–3 Data Storage and Representation

A domain is, simply, a set of values. A relation is a subset of the cartesian product of domains. Since conventional relational database systems do not allow type constructors to be included, domains contain atomic values (i.e. constants) only. We say that a relation in a relational database system is in first normal form (1NF), if its domains contain atomic values only.

Integrity constraints are those constraints which ensure that the data

manipulated into a database is accurate and consistent. Integrity constraints are discussed in [Date 1981, Ullman 1981, Date 1984, Gray 1984].

A normalization procedure will be designed to translate any relation containing structured entities into a set of equivalent relations containing atomic entities only. Moreover, it is designed to prevent update anomalies and data inconsistencies [Kent 1983].

Database systems are concerned with how data should be stored in, viewed and updated in, and retrieved from a database.

§ 2–3–4 Execution Strategy

A bottom–up proof method works by resolving the original assertions in order to produce new assertions and thereby prove the goal. Resolution in a bottom–up method uses the modus ponens inference rule (from formulas B and A ← B we can derive A). From the modus ponens inference rule we observe that bottom–up resolution cannot be carried out until all original assertions are known. The strategies for evaluating queries in relational database query languages are based on a bottom–up proof method.

§ 2–4 Discussion

§ 2–2 and § 2–3 discuss type checking and representation of data, and execution strategies for LPLs and RDBSs. In this section we discuss the relationships between LPLs and database systems.

Logic programming languages allow a programmer to model information more naturally than relational databases by using complex

facts and querying these facts using predicates containing complex arguments. In conventional relational database management systems, we are forced to normalize the relations and to use conventional relational query languages. This ability to store and query the complex facts in logic programming languages, makes them more powerful and flexible for expressing information than conventional database systems [Zaniolo 1985].

§ 2–3–1 showed that a safe expression in the tuple relational calculus is equivalent to a RAE.Tuple relational calculus and PROLOG share the following characteristics :

a- They are based on first–order predicate logic in the sense that they are built up from first order predicate logic operators.

b- They are declarative languages. They allow end–users to specify exactly the properties they require without having to specify how the process should be done.

c- They may be used to define infinite relations (q.v. § 2–3–1 and § 2–2–4).

Safety is defined as a property of a program which checks that each variable in the program is evaluated within a finite domain. Therefore, if we can prove at compile time that a program contains only variables whose type is a finite set, then the program is safe.

A universe of LPLs domain may be infinite (q.v. § 2–2–1). However, a universe of DBS domain is finite (q.v. § 2–3–2). Since a program with a Herbrand universe as domain may have infinitely many interpretations,

then the program may be unsafe. Therefore, in order to check the safety of a program we have to check whether its universe is finite or infinite.

§ 2–5 Summary

In § 2–4 we discussed the similarity between LPLs and tuple relational calculus. Moreover, § 2–3–1 showed the relationship between tuple relational calculus and relational algebraic expressions. Therefore, if we could define a similar notion of safety for PROLOG programs, then as a consequence of Ullman's theorems [Ullman 1981], we could say that: If E is a relational algebraic expression, then there is a safe rule in PROLOG equivalent to E.

LPLs are untyped languages, while RDBSs are strongly typed systems. In order to compile a logic program into a set of relational algebraic expressions we have to add type information to the program. Therefore, we have to extend a LPL to include a type system which makes it statically and strongly typed. We propose TPROLOG as such an extension to PROLOG.

Finally, LPLs allow structured terms in programs. Therefore, in order to compile a logic program into conventional relational database we have to normalize structured terms.

Chapter 3: PROLOG and Types

§ 3–1 Introduction

PROLOG is not a strongly typed language, and does not provide static type checking (q.v. § 2–1–1). However, relational algebraic databases provide static type checking. Moreover, its languages are strongly typed (q.v. § 2–3–2). Therefore, in order to compile a PROLOG program into RAEs we have to extend PROLOG to become a strongly typed language. The language proposed here is called TPROLOG.

This chapter consists of five sections. § 3–2 reviews some of the related work. § 3–3 describes the extension of PROLOG to become a typed language (i.e. TPROLOG). § 3–4 compares TPROLOG with other typed PROLOGs, while § 3–5 gives the advantages of using TPROLOG.

§ 3–2 Related Work

In this section we are, briefly, discussing the data type facilities in TURBO PROLOG and Educe*. These two systems are described in more detail in [Patrice 1987] and in [Bocca 1989].

§ 3–2–1 TURBO PROLOG

TURBO PROLOG is a PROLOG language implemented for the IBM PC [Patrice 1987]. The main differences between PROLOG and TURBO PROLOG are as follows:

1 – TURBO PROLOG is a compiled language, while PROLOG is an interpreted language.

2 – TURBO PROLOG is a strongly typed language, while PROLOG is not. Therefore, TURBO PROLOG is more restricted than PROLOG in searching solution space.

These differences make TURBO PROLOG faster than PROLOG.

A TURBO PROLOG program is divided into sections. Mainly, we are interested in the following sections:

1 – Domain declaration (domains): in this section the type (i.e. domain) of each argument name (i.e. attribute) in each clause in the program is defined. The domain of an attribute is either a basic type (i.e. integer, real, string, char, etc), a homogeneous list, or a domain that consists of compound object declared by stating a functor and the domain of all sub–arguments. For example, we may write a domain section for the PROLOG program in figure 1–2 as follows:

```
domains
    city, post_code, first_name, last_name, job, post_grad : string
    under_grad, grad_subject, school_name, school_city : string
    year,flat_no, house_no, age: integer
    home :none ;
      flat(flat_no, house_no, post_code);
      house(house_no, post_code)
    addresses : address(home, city)
    grad_school : school(school_name, school_city)
```

Note that since one of the degree types is a heterogeneous list, we cannot define its domain.

2 – Predicate declaration (predicates): In this section, a predicate is defined by its argument names (i.e. attributes). For example, in order to rewrite the PROLOG program in figure 1–2 in TURBO PROLOG we have to add the following:

```
predicates
    person(first_name, last_name, addresses, age)
    emp(first_name, last_name, degree)
    glaswegian_infant(last_name, first_name, age)
    glaswegian_emp(last_name, first_name, school, year)
```

3 - Clauses (clauses): This section contains facts and rules. For example, the PROLOG program in figure 1–2 becomes as a component of clauses section in TURBO PROLOG, when we rewrite it in TURBO PROLOG.

§ 3–2–2 Educe*

Educe is a logic programming system based on the coupling and the integrating of PROLOG and QUEL [Bocca 1986]. Moreover, in order to compute a query answer Educe translates a query, which is written in PROLOG, into QUEL. Since PROLOG is not strongly typed language, the query in Educe is untyped.

Educe* is a logic programming system which follows up from Educe [Bocca 1989]. One of the main differences between Educe and Educe* is that PROLOG in Educe* is extended to include a typing system, whilst in Educe PROLOG does not have any typing system. Moreover, PROLOG in Educe*, unlike TURBO PROLOG, does not consist of separate sections. However, different syntax is used in defining the extended PROLOG.

In addition to basic types, which are defined systematically, there are new types which may be defined by a programmer [Bocca 1989]. The new types are as follows:

1 – enumerated types. For example, in order to write the program in figure 1–2 in Educe* the following enumerated type have to be written.

```
?- adt( city, [glasgow, london, edinburgh, manchester, birmingham, reading]).
?- adt( under_grad, [hs, primary]).
?- adt( post_grad, [msc ,phd, diploma]).
?- adt( subject, [ba, computer, engl, maths, engineering, biology, medicine]).
?- adt( school_name, [glasgow_university, edinburgh_university,
        heriot_watt_university]).
?- adt( job, [guard, vp, staff, porter,]).
?- adt(post_code, [g1, g2, g3, g12, g20]).
```

2 – Fixed structure types. These are used to define the type of complex term. For example, we may define the type of complex terms in the program of figure 1–2 as follows:

```
?- adt(flat(integer, integer, string, post_code)).
?- adt(house( integer, string, post_code)).
?- adt(degree1(under_grad, integer)).
?- adt(degree2(post_grad, subject, school(school_name, city), integer)).
```

Type declarations in Educe* are syntactically similar to the type definitions. Moreover, there are some built–in predicates used for type checking. Note that no variant types are allowed. For example, we are unable to write type definitions for complex terms whose functor is address and for the heterogeneous list. In addition, since person, and emp contain terms of variant types we cannot write type declaration for them.

§ 3–3 TPROLOG

TPROLOG is a language which follows up from PROLOG. It is a strongly typed language. Moreover, it is statically typed language. A TPROLOG program consists of three sections: type definitions, facts type declarations, and facts and rules. The syntax of TPROLOG is in Appendix A. However, in this section, we are mainly interested in the typing system of the TPROLOG.

This section is divided into three sub–sections: § 3–3–1 discusses the type definitions. § 3–3–2 discusses the type checking, while § 3–3–3 discusses deducing types for variables.

§ 3–3–1 Type Definition

A type is represented as a unary relation (i.e. enumerated type) or as a type definition program [Sterling 1986]. A type definition program is a PROLOG procedure P: the corresponding type is the set of terms which satisfy P.

In the following, a type definition procedure name is used to refer both to the type definition procedure and to the corresponding type.

Types are either basic types or non–basic types. Basic types (e.g. integer, real, string) are pre–defined. Non–basic types are defined by the user as type definition programs whose syntax is given in Appendix A. Non–basic type are simple types and structured types. Note, no recursive type definition are allowed. Simple types are enumerated types and sub–types. They are defined as follows:

a) Enumerated types are, semantically, the same as the enumerated types in Educe*. However, they differ in the syntax. Here, $ f({a_1, ... ,a_n}). (where ($\forall$ i: $1 \leq i \leq n$) a_i is a ground term and f is a type–name) is the syntax form of an enumerated type. For example, city, under_grad, post_grad, subject, school–name, job, and post–code in figure 3–1 are enumerated types of the program in figure 1–2

b) Sub–types which are sub–range types of the form $ t (< t1, int1, int2>)., (where t1 is either basic type or simple type) defines t to be a sub–type of t1 ($t \subseteq t1$), such that the elements of t are the sub–range of the elements of t1 specified by int1 to int2, int1 ≤ int2, as follows:

If t1 is basic type, then int1 and int2 are interpreted as follows:

1) If t1 is string, then int1 and int2 give the minimum and maximum length of strings in t. For example, name and street in figure 3–1 are sub–types.

2) If t1 is numerical, then int1 and int2 give the smallest and largest number in t. For example, age, year, and house_no in figure 3–1 are sub–types.

If t1 is a simple type, then int1 and int2 are defined as follows:

1) If t1 is an enumerated type defined by $ t1({a_1, ... ,a_n})., then t is an enumerated type defined by $ t({a_{int1}, ... ,a_{int2}}).. Note, we must have $1 \leq int1 \leq int2 \leq n$.

2) If t1 is a sub–type defined by $t1(<t2, int3, int4>), where int3 and int4

are defined as above, then t is the sub–type effectively defined by $ t (<t2, int5, int6>)., where int5 = int1+int3-1, and int6 = int2+int4-1.

Suppose, t1, t2, and t3 are sub–types, then they satisfy the following properties:

1) If t1 $\subseteq$ t2 and t2 $\subseteq$ t3, then t1 $\subseteq$ t3.

2) if t1 = t2, then t1 $\subseteq$ t2 and t2 $\subseteq$ t1.

```
$ name(<string, 1, 10>).
$ street( <string, 1,30>).
$ city({glasgow, london, edinburgh, manchester, birmingham, reading}).
$ under_grad({hs, primary}).
$ post_grad({msc ,phd, diploma}).
$ subject({ba, computer, engl, math, engineering, biology, medicine}).
$ school_name({glasgow_university, edinburgh_university, heriot_watt_university}).
$ job({guard, vp, staff, porter,}).
$ post_code({G1, G2, G3, G12, G20}).
$ age(<integer, 0, 200>).
$ year(<integer, 1800, 2100>).
$ house_no(<integer, 1, 1000>).
$ f1(none).
$ f2 (degree1(degree_name:  under_grad, degree_year: year)).
$ f3(degree2( degree_name: post_grad, degree_subject:subject, degree_school: f4,
        degree_year:year)).
$ f4(school( name: school_name, school_city: city)).
$list1([{f2, f3}]).
$ addresses(address(house_address:home, city_address: city)).
$ flat_address(flat(flat_no: integer, building: house_no, street_name: street,
        code: post_code)).
$ house_address(house(building: house_no, street_name: street, code: post_code)).
$ qualification([f1, list1]).
$ home([f1, house_address, flat_address]).
```

Figure 3–1. Type definitions of the program in figure 1–2.

A structure type is a type of complex ground terms (i.e. complex type), a type of lists (i.e. list type), or a set of variant types (i.e. variant type).

a) A complex type is defined by a unary complex fact $ t(t1(a11: t11, ... , a1n: t1n))., where t1 is an n–ary function symbol of type $t11 \times t12 \times ... \times t1n \rightarrow t$ and $t \notin \{t11, ... , t1n\}$, and $(\forall i : 1 \leq i \leq n)$ t1i is a type name and a1i is an attribute name†. If the function symbol t1 has degree zero, then the type is the function symbol itself (i.e. $t1 \rightarrow t$). For example, the complex type definitions f1, f2, f3, and f4 of the program in figure 1–2 are shown in figure 3–1.

b) A list type is defined by a unary fact $ t([{ t1, ... , tn}]). (where t, t1, ... , tn are type names and $t \notin \{t1, ... , tn\}$); the argument in the fact is a list of type names: the corresponding type is a set of finite lists, where each element (in a list) has one of the types in the list type as its type. For example, list1 in figure–3–1 is a list type.

c) A variant type is defined by a unary fact $ t([t_1, ... , t_n]).$, where $t_1, ... , t_n$ are type names and $t \notin \{t1, ... , tn\}$. An element x is a member of t if and only if it is a member of t_i, for some i in $1, ... , n$. For example, the variant type definition qualification of the program in figure 1–2 are shown in figure 3–1.

---

† The definition excludes self–recursive types, but permits mutually recursive types. An extension which restricts the program to backward references would overcome this.

§ 3–3–2 Type Checking

The type checking of facts, rule body predicates, and goals are similar. However, we have to consider the following differences.

a) The facts must type check against only the fact declarations, whilst body predicates and goals may type check against the fact declarations or the deduced declarations (i.e. rule types). For example, the facts in the program of figure 1–2 are typed checked against the fact declarations in figure 3–2.

b) Facts contain only ground terms, whilst goals and body predicates may contain non–ground terms as well as ground terms.

c) The type checking of facts just checks the correctness of ground terms, whilst that of goal and body predicates checks the consistency between the body predicates or the sub–goals, as well as the correctness of the ground terms.

```
% emp ( first_name: name, last_name: name,job_name: job, degree: qualification).
% person( first_name: name, last_name: name, home_address: addresses, person_age: age).
```

Figure 3–2. The fact declarations of a PROLOG program in figure 1–2.

Now, we consider the type checking of terms in an atom (i.e. a fact, a body predicate, or a sub–goal) $p(x_1, \ldots, x_n)$ which corresponds to the type $\%p(a_1 :t_1, \ldots, a_n :t_n)$. Each x_i $(1 \le i \le n)$ is either a ground term or a term containing variables.

a) If x_i is a term containing variables, then the type of x_i is t_i.

b) If x_i is a ground term, then x_i must be an element of t_i (where t_i is either a single type, or an element in a variant type).

As an example of type checking, consider the following fact from figure 1–2.

emp (joe, cool, porter, none).

This fact is well typed because:

joe ∈ name,
cool ∈ name,
porter ∈ job,
none ∈ f1 ∧ ~(none ∈ list1)

Consistency checking checks that a variable V, which appears in body predicates or sub–goals, is compatible. The deduced types (q.v. § 3–3–3) of V are {t1, ... , tk} where ∀ i 1 ≤ i ≤ k the domain of ti is Ti. The consistency checking assumes that there is a unique domain T ∈ {T1, ... , Tk} where T = glb({T1, ... ,Tk}). For example, the following PROLOG program is inconsistency.

```
%p1(a1: integer).
%p2(a2: string).
r(X) :- p1(X), p2(X).
```

§ 3–3–3 Type Deduction

Type deductions deduce types for the rule head variables from the rule body predicates. It is a consequence of the type checking and consistency checking (q.v. § 3–3–2). Every variable in a rule head appears somewhere

in the body predicates.

In the following, we assume V and V' are variables with the same names, where V occurs in a rule head and V' occurs some where in the rule body predicates. The type of V is deduced as follows:

1) If V' occurs in exactly one body predicate, then the type of V is the type of V'.
2) If V' occurs in more than one body predicates (i.e. V' types are {t1, ... , tk} where $\forall$ i $1 \leq i \leq k$ the domain of ti is Ti), then the type name of V is t where the domain of t is T, and T = glb({T1, ... ,Tk}).

As an example, we give the deduced types for the following rule head variables in the program of figure 1–2.

glaswegian_infant(LN: name, FN : name, Age :age).
glaswegian_emp(Ln: name, Fn: name, Sch : f4, Yr :Year).

Moreover, a term x in a rule head may be non–variable term (i.e. either constant, complex term or list). A complex term or list may contain variables. A new type name is generated by the system for x. It is defined as follows:

1) If x is a complex term, then the corresponding type of x is defined in the same way as the complex type is defined (q.v. § 3–3–1 (a)).

2) If x is a list, then the corresponding type of x is defined in the same way as the list type is defined (q.v. § 3–3–1 (b)).

3) If x is a constant, then the corresponding type of x is defined in the

same way as the enumerated type is defined (q.v. § 3–3–1 (a)).

Note that the type of terms, which are in the structured term (i.e. complex term, or list), are deduced as shown above.

§ 3–4 Discussion

§ 3–2 discusses the typing systems in TURBO PROLOG and Educe*, while § 3–3 discusses the typing systems in TPROLOG. In this section we discuss the differences and the similarities between TURBO PROLOG, Educe*, and TPROLOG.

TURBO PROLOG does not allow any new simple types to be defined (i.e. enumerated type, and sub–range type), while Educe* and TPROLOG do. However, Educe* allows only enumerated types to be defined, while TPROLOG allows sub–range types to be defined too.

Educe* does not differentiate between type definitions (i.e. domains) and type declarations (i.e. predicates), while TPROLOG and TURBO PROLOG do. Moreover, Educe* type declarations are represented in the same way as type definitions. In Educe* all facts, which have same predicate name and arity, correspond to one type definition, while in TURBO PROLOG and TPROLOG they correspond to one type declaration. In both TPROLOG and TURBO PROLOG, type names in type declaration, may correspond to variant types. Note, TURBO PROLOG allows variant types for complex terms only. Moreover, lists in TURBO PROLOG are homogeneous, while in TPROLOG they may be heterogeneous.

Finally, Educe* allows a programmer to define the type of variables in the rule, while TURBO PROLOG allows a programmer to define the type

of arguments of a rule in predicates. However, TPROLOG deduces the types of rule variables.

§ 3–5 Summary

§ 3–4 compares different systems with TPROLOG. This comparison extracts the following advantages of using TPROLOG.

1) It enables us to write a program in a less restrictive way and closer to PROLOG.

2) It enables us to have a rich typing system.

3) It enables us to get the benefits of PROLOG flexibility in representing information as well as the benefits of strongly typed programming languages.

Chapter 4: Safety

§ 4–1 Introduction

In chapter 2, we reviewed the execution strategies in LPLs and RDBSs, and concluded that there is a need for checking the safety of the execution of the queries. In this chapter, we discuss methods used for safety checking.

The outline of this chapter is as follows: § 4–2 discusses safety checking at execution time, whilst § 4–3 discusses safety checking at compile time. § 4-4 compares our approach to safety checking with others, and § 4–5 summaries the advantages of using our approach for safety checking.

§ 4–2 Safety Checking at Execution Time

§ 4–2–1 Safety for Non–Recursive PROLOG Programs

A notion of safety for non–recursive PROLOG programs is introduced by Zaniolo [Zaniolo 1985, Zaniolo 1986], which is similar to the notion of safety introduced by Ullman [Ullman 1981]. Zaniolo's approach is based on the notion of a magic basis of a variable: a technique which uses the notion of a proof procedure using connection graphs (PCG) from [Kowalski 1975]. (A PCG is a graph, which represents all possible paths of the resolution (e.g. top–down or bottom–up) of a set of first order predicate clauses).

In the following, we refer to a graph which represents the structure of the goal as a goal tree, and a graph which represents the structure of a rule

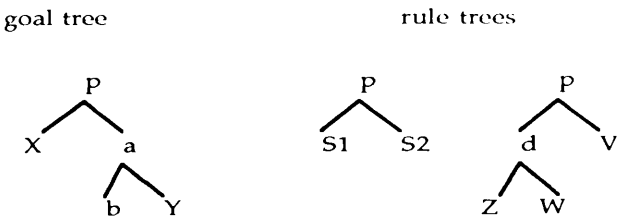
head as a rule tree, where the address (i.e. root) of these trees is either a rule head predicate name or a goal predicate name.

The magic basis of a variable in a query (or sub–goal) is a mapping between variables in the goal tree (represented by a PCG) derived from the query, and terms in the rule trees of the rules which unify with the goal. For a given variable X in a goal tree, the magic basis of the variable X is given by the union of P(X) and LP(X), where P and LP are defined as follows: Let p1 and p2 be atoms with the same address (i.e. p1 and p2 have the same predicate name), let X be a variable occurring in p1 and t be a term occurring in p2, where X and t are in a same parameter position.

a) If X and t are in a same position in their respective trees (i.e. they are at a same level), then t is a partner of X and so $t \in P(X)$. There are no other elements in P(X).

b) If X and t are not at the same level, but X has at least one ancestor which is partner of t (i.e. X is lower than t), and t is a variable, then t is a lower partner of X and so $t \in LP(X)$. There are no other elements in LP(X).

For example, if we have the following trees:



then, $d(Z,W) \in P(X)$, $S1 \in P(X)$, $S2 \in LP(Y)$, and $V \in LP(Y)$.

In general, the safety of a query depends on the safety of its variables, where the safety of a variable of a query depends on whether the magic basis of the variable is finite or not, and on the safety of the variables or constants with which it is unified.

More precisely, the safety rules which govern rule/goals by using the magic basis of a variable are as follows :

Rule 1) Every variable in a goal which unifies with a database relation (i.e. only facts) is safe.

Rule 2) Every constant is safe.

Rule 3) If a variable X in a rule is safe, then all variables in P(X) or LP(X) of a goal, which are unified with the rule, are safe.

Rule 4) If a variable X in a goal is safe, then all variables in P(X) or LP(X) of a rule, which are unified with the goal, are safe.

Rule 5) If all variables in an arithmetic exp are safe then the variable V in V is exp is safe.

§ 4–2–2 Safety for Recursive Datalog Programs

The datalog programming language [Krishnamurthy 1988] is a logic programming language based on first–order predicate logic. Although it is similar to pure PROLOG (i.e. PROLOG as a first–order logic without any built in predicates such as cut), it does not allow function symbols (i.e. simple terms).

A notion of safety for recursive datalog has been investigated by many researchers [Ullman 1985, Tsur 1986, Ramakrishnan 1987, Bancilhon 1985]. Ullman's approach uses a rule/goal graph to check the safety of the execution of a datalog query. The rule/goal graph is a technique which also uses the notion of a PCG [Kowalski 1975]. The safety conditions, which will be explained later, are applied after mapping the query into node(s) in the rule/goal graph of the unified rule(s).

A rule/goal graph consists of nodes and arcs. Nodes represent all possibilities of representing the variables in an atom as free (f) (i.e. those variables which are substituted by the unified goal variables) or bound (b) (i.e. those variables which are substituted by the unified goal constants), while arcs represent the dependency relationship between the atoms in a rule. Since logic programming languages do not differentiate between free variables and bound variables in the sense above, every variable is potentially both a free variable and a bound variable. Also, each variable of the rule head will be called an input variable or an output variable as explained below. If a rule head variable occurs only as an operand to an arithmetic atom or comparison atom (with one exception), then that variable is called an input variable. The exception is: if the rule head variable is V and the atom is V is exp (for any expression exp), then V is not an input variable. If a rule head variable is not an input variable then it is an output variable (we note that the rule/goal graph is generated at compile time, so an output variable may become an input variable at execution time).

In the following, we refer to an atom which can be unified only with a fact as a fact atom and otherwise we refer to it as a non-fact atom.

The node associated with an atom in a rule body, which represents a fact atom, is defined as follows : if F is an atom containing n variables, then the node associated with F is $F^{x_1 \cdots x_n}$, where $(\forall g{:}1 \leq g \leq n)\ x_g = b$; all variables are bound. The nodes associated with a non–fact atom are generated as follows :

a) In a non–recursive program the nodes are addressed by their predicate symbol name. In a recursive program, for sake of simplicity and non–ambiguity, the rule head is addressed by a new name. When a rule head p is addressed by q then we refer to q as the node name associated with p. For example, in the program of figure 1–1 we may address the rule head with predicate symbol factorial by r1.

b) The number of nodes associated with an atom is determined by the number of variables in it. Thus, there are 2^n nodes, where n is the number of variables in that atom.

c) Nodes are defined as follows: every variable in an atom can be either f or b, thus if F is an atom containing n variables, then there are 2^n nodes of the form $F^{x_1 \cdots x_n}$, where $(\forall g : 1 \leq g \leq n)\ x_g \in \{b,f\}$.

Arcs are generated as follows:

a) For each body predicate $F(A_1, \ldots, A_n)$ and rule head $G(B_1, \ldots, B_m)$ which contains output variables, where $A_1, \ldots, A_n$ are simple terms and $B_1, \ldots, B_m$ are simple terms, such that for some k, where $1 \leq k \leq n$, and some h, $1 \leq h \leq m$, B_h is output variable, and A_k and B_h are the same variable, there are arcs defined as follows:

1) If $F(A_1, \ldots, A_n)$ is a non–fact atom, then the arcs are from

$$F^{x_1 \ldots x_{k-1} b x_{k+1} \ldots x_n} \text{ to } r(G)^{y_1 \ldots y_{h-1} b y_{h+1} \ldots y_m}$$

and from each node of form

$$F^{x_1 \ldots x_{k-1} f x_{k+1} \ldots x_n} \text{ to } r(G)^{y_1 \ldots y_{h-1} f y_{h+1} \ldots y_m}$$

2) If $F(A_1, \ldots, A_n)$ is a fact atom, then the arcs are from

$$F^{z_1 \ldots z_{k-1} b z_{k+1} \ldots z_n} \text{ to } r(G)^{y_1 \ldots y_{h-1} b y_{h+1} \ldots y_m}$$

and from each node of the form

$$F^{z_1 \ldots z_{k-1} f z_{k+1} \ldots z_n} \text{ to } r(G)^{y_1 \ldots y_{h-1} f y_{h+1} \ldots y_m}$$

where r(G) is the node name associated with G (since addressing may have been carried out, as described above), and

$(\forall\, h : \{1, \ldots, k-1, k+1, \ldots, n\})\ z_h = b,$

$(\forall\, g : \{1, \ldots, k-1, k+1, \ldots, n\})\ x_g \in \{b,f\}$, and

$(\forall\, k : \{1, \ldots, h-1, h+1, \ldots, m\})\ y_k \in \{b,f\}.$

b) If a body predicate $F1(A_1, \ldots, A_n)$ occurs before (i.e. to the left of) a body predicate $F2(B_1, \ldots, B_m)$, where $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are simple terms, such that for some k, where $1 \leq k \leq n$, and some h, where $1 \leq h \leq m$, A_k and B_h are the same variables, then there is an arc from each node of the form

$$F1^{x_1 \ldots x_{k-1} b x_{k+1} \ldots x_n} \text{ to } F2^{y_1 \ldots y_{h-1} b y_{h+1} \ldots y_m}$$

and from each node of the form

$$F1^{x_1 \cdots x_{k-1} f x_{k+1} \cdots x_n} \text{ to } F2^{y_1 \cdots y_{h-1} f y_{h+1} \cdots y_m}$$

where $(\forall\, g : \{1, \ldots, k-1, k+1 \ldots, n\})\ x_g \in \{b,f\}$,
and $(\forall\, k : \{1, \ldots, h-1, h+1, \ldots, m\})\ y_k \in \{b,f\}$.

c) For each body predicate $F(A_1, \ldots, A_n)$ and a rule head $G(B_1, \ldots, B_m)$ which contains input variables, where $A_1, \ldots, A_n, B_1, \ldots, B_m$ are simple terms, such that for some k, where $1 \le k \le n$, and some h, where $1 \le h \le m$, B_h is an input variable, A_k and B_h are the same variables, there is an arc form each node of the form

$$r(G)^{y_1 \cdots y_{h-1} b y_{h+1} \cdots y_m} \text{ to } F^{x_1 \cdots x_{k-1} b x_{k+1} \cdots x_n}$$

and from each node of the form

$$r(G)^{y_1 \cdots y_{h-1} f y_{h+1} \cdots y_m} \text{ to } F^{x_1 \cdots x_{k-1} f x_{k+1} \cdots x_n}$$

where r(G) is the node name associated with G (since renaming may have been carried out, as described above),
$(\forall\, g : \{1, \ldots, k-1, k+1 \ldots, n\})\ x_g \in \{b,f\}$, and
$(\forall\, k : \{1, \ldots, h-1, h+1, \ldots, m\})\ y_k \in \{b,f\}$.

d) Finally, arcs are also directed from a non–recursive rule head to a body of another rule, or from a recursive rule head to its respective body predicate, when the atom is unified with that rule. These arcs are defined as follows: for each atom in a rule body of the form $F(A_1, \ldots, A_n)$ and rule head $F(B_1, \ldots, B_n)$, where $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ are simple terms, there is an arc from each node of the form

$r(F)^{x_1 \cdots x_n}$ to $F^{x_1 \cdots x_n}$

where r(F) is the node name associated with F (since renaming may have been carried out, as described above), and ($\forall\ g : 1 \leq g \leq n$) $x_g \in \{b,f\}$.

An example of the generated rule/goal graph is the adjacency matrix of the generated rule/goal graph in figure 4–1 of the program in figure 1–1. Note, the rule head is addressed by r1.

from \ to	factorial^{bb}	factorial^{bf}	factorial^{fb}	factorial^{ff}	$\_^{bbb}$	$\_^{bfb}$	$\_^{fbb}$	$\_^{ffb}$	$*^{bbb}$	$*^{bbf}$	$*^{bfb}$	$*^{bff}$	$*^{fbb}$	$*^{fbf}$	$*^{ffb}$	$*^{fff}$	$r1^{bb}$	$r1^{bf}$	$r1^{fb}$	$r1^{ff}$
factorial^{bb}	1	1	1	1					1		1		1		1					
factorial^{bf}										1		1		1		1				
factorial^{fb}									1		1		1		1					
factorial^{ff}										1		1		1		1				
$\_^{bbb}$	1	1							1	1			1	1						
$\_^{bfb}$	1	1									1	1			1	1				
$\_^{fbb}$			1	1					1	1			1	1						
$\_^{ffb}$			1	1							1	1			1	1				
$*^{bbb}$																	1		1	
$*^{bbf}$																	1		1	
$*^{bfb}$																	1		1	
$*^{bff}$																	1		1	
$*^{fbb}$																		1		1
$*^{fbf}$																		1		1
$*^{ffb}$																		1		1
$*^{fff}$																		1		1
$r1^{bb}$	1				1		1		1	1			1	1						
$r1^{bf}$		1			1		1		1	1			1	1						
$r1^{fb}$			1			1		1			1	1			1	1				
$r1^{ff}$				1		1		1			1	1					1	1		

Figure 4-1. Adjacency matrix of the program in figure 1-1.

Note, we adopt the following convention: given an atom P, unless we state explicitly otherwise, we will use the notation node P to refer to all superscripted nodes P in the graph.

In general, the safety of a query at execution time depends on whether the query adopts a safe execution paths (i.e. the goal graph) in the rule/goal graph of the rule(s) which are unified with it. Therefore, the goal graph is a sub–graph of the program rule/goal graph which represents all possible execution paths of the query. For example, the goal graph of the query ?– factorial(X, Y) is graph of the whole program in figure 1–1, whilst the goal graph of the query ?–factorial(3, X) is as shown in figure 4–2.

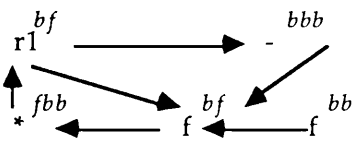


Figure 4–2. The goal graph of the query ?–factorial(3,X).

We define an execution path of node P to be a path whose source is a node with in–degree 0 and target node P. A safe execution path of node P is a path whose source node is a fact node.

Consequently, a safe execution path of a node is a path through nodes which are classified as safe by the following rules: (We note that the rules refer only to nodes and edges in the given path)

1) If N is a node of a rule head, and P is a safe node of a body atom of that rule, and there is an arc from P to N, then N is safe.

2) If P is a node of a rule body atom which is unified with a fact atom, then P is safe.

3) If P' is a node of a rule head, P is a non–recursive body atom, and there is an an arc from P' to P and P' is safe, then P is safe.

4) If P is an arithmetic expression atom node and all variables on the right hand side of the associated predicate are represented as *b*, then P is safe.

5) If P is a comparison atom node and all variables of the associated atom are represented as *b*, then P is safe.

6) If P is a recursive body atom node, P1 is a safe atom, and there is an arc from P1 to P, then P is safe.

For example, the adjacency matrix in figure 4–1 is the goal `?-factorial(X,Y)` graph. It contains at least an unsafe path (i.e. $\text{factorial}^{ff} \rightarrow {}^{*bff} \rightarrow \text{r1}^{bb}$). Therefore, the query is unsafe. However, the goal ?–factorial(3,Y) is safe, because all paths in the goal graph of figure 4–2 are safe.

§ 4–3 Safety Checking of a PROLOG Program at Compile Time

In the last section we defined the notion of safety, by using the notion of magic basis of a variable and by using the notion of rule/goal graph. Both rule/goal graph and magic basis of a variable are used to define the safety of a logic program at execution time: safety conditions are used to check the safety of an atom in the rule/goal graph approach, while they are

used to check the safety of a variable in the magic basis of a variable. However, in practice, these approaches are equivalent for non–recursive datalog programs.

In this section, we extend the safety checking procedure for non–recursive PROLOG programs into a safety checking procedure for recursive PROLOG programs by integrating the notion of the rule/goal graph with the notion of the magic basis of a variable. Moreover, we apply the procedure at compile time instead of at execution time.

For sake of simplicity, in the following, we represent a complex term as *b* (bound), if each variable or complex term in it is represented as *b* and we represent it as *f* (free), if each variable or complex term in it is represented as *f*.

Our graph, like the rule/goal graph of Ullman, consists of arcs and nodes. The generation of nodes for our graph is similar to the generation of nodes for rule/goal graph with the following three exceptions.

First, the main difference is that fact body atoms and non–fact body atoms are differentiated, whereas in our graph they are not.

Second, a fact is not represented in the rule/goal graph, while in our graph it is represented by a special node. Note, nodes represent the constants in an atom as *c* (constant). The (special) node, which represents a fact formula, is constructed in the following way: if $F(a_1, \ldots, a_n)$ is a fact formula, then the (special) node associated with it is $F^{x_1, \ldots, x_n}$ (where $(\forall g{:}1 \leq g \leq n)\ x_g$ is the term derived from a_g by replacing all occurrences of constants (in a_g) by the term c). For example, if $F(a_1, \ldots, a_{k-1}, f1(a'_1, \ldots, a'_m), a_{k+1}, \ldots, a_n)$ is a fact formula with constants

$a_1, \ldots, a_{k-1}, a'_1, \ldots a'_m, a_{k+1}, \ldots, a_n$, then the node associated with it is $F^{x_1 \ldots, x_{k-1} f1(y_1 \ldots, y_m), x_{k+1} \ldots, x_n}$ where $(\forall g : 1 \leq g \leq m)\ y_g = c$, and $(\forall h : \{1, \ldots, k-1, k+1, \ldots, n\})\ x_h = c$.

Third, we incorporate information about the constants and function symbols occurring in atoms into the associated node names. For example if $F(T_1, \ldots, T_n)$ is an atom (where $T_1, \ldots, T_n$ are terms and), then it is represented by a set of nodes where each node is of the form $F^{x_1 \ldots, x_n}$, where $(\forall g : 1 \leq g \leq n)$

a) If T_g is a constant, then $x_g = c$,

or

b) If T_g is a variable, then $x_g \in \{b,f\}$,

or

c) If T_g is a complex term, then x_g is a term derived from T_g by replacing all occurrences of non–complex terms in T_g by the terms c, b, or f.

The generation of arcs in our graph is similar to the generation of arcs in the rule/goal graph. However, since the magic basis of a variable defines the relationship between a goal or a sub–goal and the unified rule head, the generation of arcs (in our graph) directed to an atom in a (non–recursive) rule body from another rule head or a fact, or to a (recursive) rule body form its rule head is extended as follows:

a) If $F(V_1, \ldots, V_{k-1}, D, V_{k+1}, \ldots, V_n)$ is an atom in a rule body and $F'(A_1, \ldots, A_{k-1}, B, A_{k+1}, \ldots, A_n)$ is a rule head or fact (where $(\forall g : \{1, \ldots, k-1, k+1, \ldots, n\})\ V_g = A_g$, V_g and A_g are terms, D is a variable, B is a complex term, and F, F' are the same predicate name), then using the definition of $B \in P(D)$ from the magic basis of a variable

(q.v. § 4–2–1) there are arcs defined as follows:

1) If F' is a rule head , then there is an arc from each node of the form

$$r(F')^{x_1 \cdots x_{k-1} b x_{k+1} \cdots x_n} \text{ to } F^{x_1 \cdots x_{k-1} b x_{k+1} \cdots x_n}$$

and from each form

$$r(F')^{x_1 \cdots x_{k-1} f x_{k+1} \cdots x_n} \text{ to } F^{x_1 \cdots x_{k-1} f x_{k+1} \cdots x_n}$$

where r(F') is the node name associated with F' (since renaming may have been carried out, as described in § 4–2–2), and $(\forall g : \{1, \ldots, k-1, k+1 \ldots, n\})\ x_g \in \{b,f\}$.

2) If F' is a fact, then there is an arc from a fact node (i.e. special node) of the form

$$F'^{z_1, \ldots, z_n} \text{ to } F^{x_1, \ldots, x_n}$$

where $(\forall g : 1 \le g \le n)$

$(z_g = c \wedge (x_g \text{ is a ground term} \Rightarrow x_g = c)$

$\wedge (x_g \text{ is not a ground term} \Rightarrow x_g \in \{b,f\}))$.

b) If $F(A_1, \ldots, A_{k-1}, f1(B_1, \ldots, B_m), A_{k+1}, \ldots, A_n)$ is an atom in a rule body and $F'(V_1, \ldots, V_{k-1}, D, V_{k+1}, \ldots, V_n)$ is a rule head (where $(\forall g : \{1, \ldots, k-1, k+1, \ldots, n\})\ V_g = A_g$, V_g and A_g are terms, D is a variable , $B_1, \ldots, B_m$ are terms, and F, F' are the same predicate name), then using the definition of $D \in LP(B_1), \ldots, D \in LP(B_m)$ from the magic basis of a variable (q.v. § 4–2–1) there is an arc from each node of the form

$$r(F')^{x_1 \cdots x_{k-1}\, b\, x_{k+1} \cdots x_n} \text{ to } F^{x_1 \cdots x_{k-1}\, f\, (y_1 \cdots y_m)\, x_{k+1} \cdots x_n}$$

and from each form

$$r(F')^{x_1 \cdots x_{k-1}\, f\, x_{k+1} \cdots x_n} \text{ to } F^{x_1 \cdots x_{k-1}\, f\, (w_1 \cdots w_m)\, x_{k+1} \cdots x_n}$$

where r(F') is the node name associated with F',

$(\forall g : \{1, \ldots, k-1, k+1, \ldots, n\})$

$x_g \in \{b,f\}$, and $(\forall i : 1 \le i \le m)\ y_g \in \{b,c\}$ and $w_g \in \{c,f\}$.

For example, the graph in figure 4–3 is generated by the above definition to represent the following program. Note, le(X, Y) is unified with a finite set of facts.

```
r1) order(nil).
r2) order(cons(X,nil)) :-.
r3) order(cons(X, cons(Y,Z))) :- le(X,Y), order(cons(Y,Z)).
```

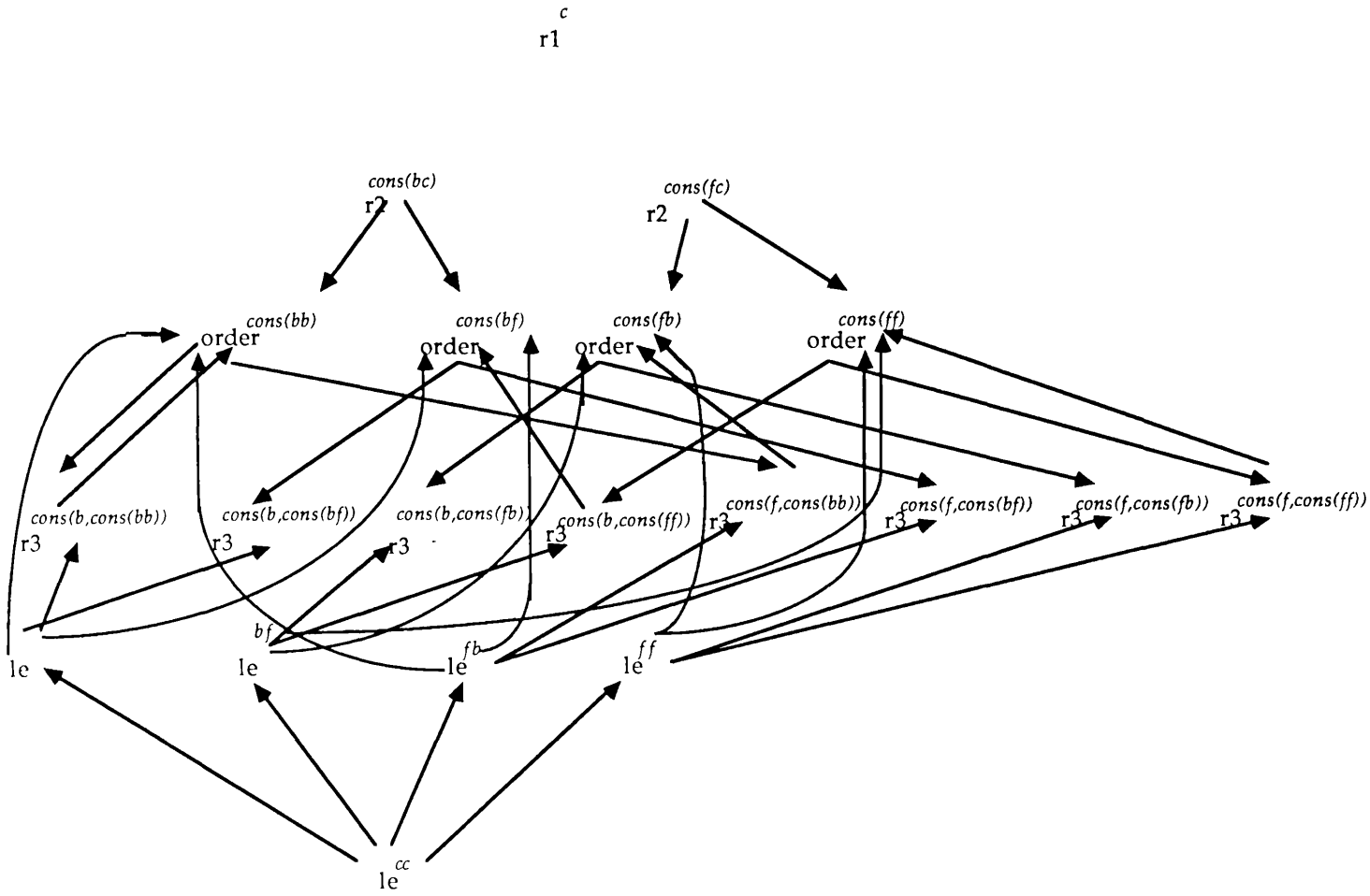


Figure 4–3. The graph of the order program

We may use the notion of safety given in § 4–2–2, to define the safety of a query mapped into this graph. However, the rules in the definition of § 4–2–2 are for checking the safety of a program at execution time, whilst we look for rules to check the safety of a program at compile time. There are no general rules to check the safety of a program at compile time (i.e. safety checking for horn clauses is undecidable [Zaniolo 1986]). As a consequence, let us consider an extent to which the safety checking algorithm given in § 4–2–2 can check the safety at compile time.

In order to do so, we define a set of rules (i.e. a PROLOG procedure where the predicate name of each rule head in the set are the same) to be either strongly safe or weakly safe with respect to their execution paths in a graph. We say that a procedure is strongly safe if all its execution paths in a graph satisfy the safety rules in § 4–2–2. Otherwise, it is weakly safe. For example, the procedure order, which is represented by the graph in figure 4–3, is weakly safe, because although are some execution paths are safe, others are unsafe. For example, the execution of r2, $r2^{bc}$ is safe. However, another execution path for r2, $r2^{fc}$ is unsafe by rule 2.

If a procedure is strongly safe, then the procedure is abstractly safe in the sense that its behaviour does not depend on the environment in which it is executed. So, we can guarantee the run time safety of a strongly safe procedure at compile time. However, if the procedure is weakly safe, then it is only safe for some execution paths but not for all. Thus, we cannot guarantee the run time safety of a weakly safe procedure at compile time.

Thus, any strongly safe PROLOG procedure has an equivalent set of RAEs. Therefore the PROLOG procedure may be compiled using one of

techniques given in [Reiter 1978, Henschen 1984, Chang 1986]. A weakly safe PROLOG procedure does not necessarily have an equivalent set of RAEs. As a consequence, we may compile (i.e. translate syntactically) such a procedure (using one of techniques described above), but the resulting RAEs may only become meaningful at execution time (if at all).

§ 4–4 Discussion

§ 4–2 discusses the safety checking at execution time, while § 4–3 discusses the safety checking at compile time In this section we compare our approach for safety checking (q.v. § 4–3) with others (q.v. § 4–2).

In order to make a comparison with Ullman's approach, we must assume that his approach is applied at compile time (instead of execution time). The rule/goal graph (q.v. § 4–2–2) and our graph (q.v. § 4–3) of the example in figure 1–1 are given in figures 4–1 and figure 4–4 respectively.

from \ to	factorial^{bb}	factorial^{bf}	factorial^{fb}	factorial^{ff}	$_{-}^{bbb}$	$_{-}^{bfb}$	$_{-}^{fbb}$	$_{-}^{ffb}$	$_{*}^{bbb}$	$_{*}^{bbf}$	$_{*}^{bfb}$	$_{*}^{bff}$	$_{*}^{fbb}$	$_{*}^{fbf}$	$_{*}^{ffb}$	$_{*}^{fff}$	$\mathrm{r1}^{bb}$	$\mathrm{r1}^{bf}$	$\mathrm{r1}^{fb}$	$\mathrm{r1}^{ff}$	factorial$^{\infty}$
factorial^{bb}									1		1		1		1						
factorial^{bf}										1		1		1		1					
factorial^{fb}									1		1		1		1						
factorial^{ff}										1		1		1		1					
$_{-}^{bbb}$	1	1							1	1			1	1							
$_{-}^{bfb}$	1	1									1	1			1	1					
$_{-}^{fbb}$			1	1					1	1			1	1							
$_{-}^{ffb}$			1	1							1	1			1	1					
$_{*}^{bbb}$																	1		1		
$_{*}^{bbf}$																	1		1		
$_{*}^{bfb}$																	1		1		
$_{*}^{bff}$																	1		1		
$_{*}^{fbb}$																		1		1	
$_{*}^{fbf}$																		1		1	
$_{*}^{ffb}$																		1		1	
$_{*}^{fff}$																		1		1	
$\mathrm{r1}^{bb}$	1				1		1		1	1			1	1							
$\mathrm{r1}^{bf}$		1			1		1		1	1			1	1							
$\mathrm{r1}^{fb}$			1			1		1			1	1			1	1					
$\mathrm{r1}^{ff}$				1		1		1			1	1					1	1			
factorial$^{\infty}$	1	1	1	1																	

Figure 4–4 Adjacency matrix for our graph of the program in figure 1–1.

By analyzing the graphs in figure 4–1 and figure 4–4 and using the node safety definition (q.v. § 4–2–2), we find the following limitations of the rule/goal graph:

a) Even though the factorial predicate is either unified with a rule or a fact, the rule/goal graph is unable to distinguish these two possibilities. As a consequence, there are no nodes with in–degree 0 and therefore there are no executions paths associated with the nodes in the graph. Thus, we can conclude nothing about the compile time behaviour of the factorial procedure.

b) Consider the graph in figure 4–4 now: There is a both safe execution path to $r1^{bb}$ and an unsafe execution path to $r1^{bb}$. The unsafe execution path is $factorial^{cc} \rightarrow factorial^{ff} \rightarrow *^{bff} \rightarrow r1^{bb}$. Since $*^{bff}$ unsafe by safe rule 4 (q.v. § 4–2–2), $r1^{bb}$ becomes unsafe by rule 1. The safe path to $r1^{bb}$ is $factorial^{cc} \rightarrow factorial^{bb} \rightarrow *^{bbb} \rightarrow r1^{bb}$. Since $*^{bbb}$ is safe by rule 4, the $r1^{bb}$ becomes safe by rule 1.

Ullman's approach would force us to conclude that the factorial procedure is unsafe, whilst in our approach we would conclude that the procedure is weakly safe. Therefore, our approach allows us, effectively, to check the safety of a logic program at compile time, if it is strongly safe. Moreover, we can also check the safety of a logic program at execution time, if it is weakly safe.

In comparison, Ullman's approach [Ullman 1985], and Zaniolo's approach [Zaniolo 1986] check the safety of logic programs at execution time only. Moreover, Krishnamurthy's approach [Krishnamurthy 1988] checks the safety of logic programs at compile time, but does not distinguish between strong safety and weak safety.

Finally, our approach handles the (pure) PROLOG language (including arithmetic expressions), whereas Ullman's approach and Krishnamurthy's approach exclude function symbols, and Zaniolo's approach excludes recursive programs.

§ 4–5 Summary

§ 4–4 compares our approach (q.v. 4–3) with others. This section summarizes the advantages of our approach

1) The rule/goal graph has been extended to include the complex terms of pure PROLOG programs, and to differentiate between rules and facts.

2) The compile time safety checking procedure has become more sophisticated in the following ways:

a) If all execution paths of the procedure P are unsafe, then P is unsafe and it should not be compiled.

b) If all execution paths of the procedure P are safe, then P is strongly safe and the compilation of it should be completed.

c) If some execution paths of the procedure P are safe and the other are not, then P is weakly safe and, although the syntactic translation to relational algebraic expressions is done at compile time, there will be more safety checking carried out at execution time.

Chapter 5: Normalizing TPROLOG programs

§ 5–1 Introduction

This chapter discusses our method of translating (i.e. normalizing) a logic program containing non–flat clauses into an equivalent logic program which contain flat clauses only. Since we adopt TPROLOG, we translate a TPROLOG program into an equivalent TPROLOG program which contain flat clauses only.

The chapter consists of five sections. § 5–2 gives a method of normalizing facts, whilst § 5–3 gives a method of normalizing rules. § 5–4 discusses goal normalization. § 5–5 compares our approach with others.

§ 5–2 Fact Base Normalization

In § 3–3 we extended the standard PROLOG language to include data types (i.e. TPROLOG). This extension, from a data base point of view, may be used as a data base schema [Atkinson 1987]. It should be used to prevent an operation from being applied to a value of an inappropriate type: any querying of the contents of a database should be done with respect to its schema (i.e. querying a TPROLOG program is done with respect to the fact declarations).

A relation in a relational database is a relation containing atomic entities only. Normalization in RDBSs transforms a schema from 1NF to higher normal form (i.e. 2NF, 3NF, etc) with respect to the functional dependencies between attributes in the relation. LPLs may contain non

atomic relations. In order to transform a logic program into RAEs, we have to flatten all structured data. Hereafter, we call this transformation normalization.

Our normalization is a procedure which flattens all structured data (from relational point of view non–structured values) in a TPROLOG program. In other words, normalization removes non–flat clauses and replaces them by flat clauses. The replacement of structured data requires the replacement of the structured data types as well. A structured data type is either a list or complex term. Moreover, we may consider a type which consists of a set of types (i.e. a variant type) as a structured data type as well. § 5–2–1 gives methods to normalize fact, whilst in § 5–2–2 discusses how fact data declarations are normalized in a similar way.

§ 5–2–1 Normalizing Facts

A structured fact is a fact where at least one of its components is a complex term, a list, or a term of variant type. In the following sub–sections we discuss the normalization of these three cases.

§ 5–2-1-1 Normalizing Complex Terms

In this section we give a method which eliminates complex terms from a non–flat fact. The method consists of introducing the existential quantifier (∃), and using skolemization which removes the quantification [Bundy 1983].

Suppose, we have the following fact f containing one complex term:

$$f(x_1, \ldots, x_{k-1}, f1(xf1_1, \ldots, xf1_m), x_{k+1}, \ldots, x_n).$$

Then, we remove the complex term $f1(xf1_1, \ldots, xf1_m)$ in f by replacing f by the following two flat facts.

$f(x_1, \ldots, x_{k-1}, c, x_{k+1}, \ldots, x_n)$.
$rf1(xf1_1, \ldots, xf1_m, c)$.

where c is a new constant of skolem type. Note that the domain of skolem type is defined systematically (i.e. every constant introduced by skolemization is a member of this type).

Note, the above example contains one complex term; we can easily generalize the method to several complex terms.

§ 5–2–1–2 Normalizing Lists

In TPROLOG, the differences between complex term and list (q.v. § 3–3) are as follows:

a) Every term of complex term has a fixed arity (i.e. a fixed number of sub–terms), whilst a term of a list type does not.

b) The type of each sub–term in a list has variant type, whilst each sub–term in a complex term does not.

Consequently, the normalization of a fact
$f(x_1, \ldots, x_{k-1}, [lx_1, \ldots, lx_n], x_{k+1}, \ldots, x_m)$., where $[lx_1, \ldots, lx_n]$ has type $list1[\{t_1, \ldots, t_g\}]$, is as follows:

1) f is transformed into
$f(x_1, \ldots, x_{k-1}, list1([lx_1, \ldots, lx_n]), x_{k+1}, \ldots, x_m)$.

2) The normalization of the fact in (1) above is done in the same way as in § 5–2–1–1:

$f(x_1, ..., x_{k-1}, c, x_{k+1}, ..., x_m)$.

$rlist1([lx_1, ..., lx_n], c)$

3) is transformed into n facts of the form

$rlist1(lx_i, c)$., for each i = 1, ..., n, and the type of lx_i is one of variant types $\{t_1, ..., t_s\}$.

4) The normalization of the resulted facts from (3) is done in the same way as the normalization process in § 5–2–1–3 below.

§ 5–2–1–3 Normalizing Terms of Variant Types

Each term of variant data type belongs to exactly one type in the variant type (q.v. § 3–3). The normalization of terms with variant type is done in two steps as follows:

step 1- The generation of complex terms: if a term x belongs to a simple type t in a variant data type, then x is transformed into a new unary complex term . The unary complex term is defined with respect to the type definition of t as follows: suppose that $[t_1, ..., t_n]$ is a variant type, $t \in \{t_1, ..., t_n\}$, and t is a simple type, then the new type for t is f(f1(t)) where f is a new type name, f1 is a new function symbol, and t : f. As a result of transforming t type definition into f1(t), x is transformed into f1(x). For example, given $ identification([social–no, full–name]). , where full–name is a complex term type (note, there is no need to transform full–name into a complex term type) and, social–no is a simple type defined by

$social-no(<integer,1, 10000>). The type name social-no in identification is transformed into f(f1(social-no)). Moreover, x is transformed into a unary complex term f1(x), which corresponds to the unary complex term data type.

step 2- The normalization of the generated complex term is as shown in § 5–2–1–1.

An example of facts normalization is that for given facts in the program of figure 1–2, the normalized form of them are in figure 5–1.

```
person(joe, cool, c1, 20).
person(max, fax, c3, 40).
person(joe,doe, c5, 3).
raddress(c2, glasgow, c1).
raddress(c4, glasgow, c3).
raddress(c6,glasgow, c5).
rnone(c2).
rflat(2l, 18, windsor_street, g20, c4).
rhouse(31, kew_drive, g12, c6).
emp(joe, cool, porter, c7).
emp(max, fax, guard, c8).
emp(fred, red, staff, c9).
rnone(c7).
rlist1(c11, c9).
rlist1(c12, c9).
rlist1(c13, c9).
rlist1(c14, c8).
rdegree1(hs, 1968, c14).
rdegree1(hs, 1975, c11).
rdegree2(msc, ba, c10, 1980, c12).
rdegree2(phd, ba, c15, 1983, c13).
rschool(glasgow_university, glasgow, c15).
rschool(glasgow_university, glasgow, c10).
```

Figure 5–1. The normalization of facts in figure 1–2

§ 5–2–2 Normalizing Fact Declarations

Any changes in facts requires changes in their declarations. Therefore, fact declaration normalization is very similar to the normalization of their respective facts. Since, in § 5–2–1, we have defined structured facts by three categories, the structured fact declarations are defined by the same categories.

The normalization of structured fact declarations is defined as follows: Suppose t is a structured data type in a fact declaration

a) If t is a complex term data type, then the fact declaration is normalized in the same way as its corresponding fact (q.v. § 5–2–1–1). Note that, the replaced attribute name has type skolem.

b) If t is a list type, then the fact declaration is normalized as follows:

1) The list data type is transformed into a unary complex term type (q.v. § 5–2–1–2) and the type of the function symbol (i.e. a list name) is a variant type of the set of list types. For example, the list type $list1([{f2, f3}])., in figure 3–1 is transformed into list1([f2, f3]), where the type of list1 is {f2, f3}.

2) The normalization of the generated complex term type is explained in § 5–2–1–2.

c) If t is a variant type, then we may consider the fact declaration as several fact declarations. However, from a relational data base point of view several data declarations for one relation (i.e. fact base) is prohibited. So, in order to normalize fact declarations, we have to eliminate several data declarations and replace them by one fact declaration. The elimination and the replacement of several fact data declarations is similar to its respective fact normalization (q.v. § 5–2–1–3). However, we have to consider the following:

1) A variant type consists of a list of types, whilst a term, which

belongs the variant type, belongs to one element in the variant type.

2) all variant types are replaced by one constant (i.e. type name). The normalization process of the variant type is similar to the normalization process in§ 5–2–1–2.

As an example, we normalize the fact declarations in figure 3–2 below:

```
% emp ( first_name: name, last_name: name,job_name: job, a1: skolem).
% person( first_name: name, last_name: name, a3: skolem, person_age: age).
%rnone(a1: skolem).
%rdegree1(degree_name: under_grad, degree_year: year, a4: skolem).
%rlist1(a4: skolem, a1: skolem).
%rdegree2( degree_name: post_grad, degree_subject: subject, a2: skolem, year, a4: skolem).
%rschool( name: school_name, school_city: city, a2: skolem).
%raddress( a1: skolem, city_address: city, a3: skolem).
%rflat(flat_no: integer, building: house_no, street_name: street, code: post_code,
        a1: skolem).
%rhouse(building: house_no, street_name: street, code: post_code, a1: skolem).
```

Figure 5–2. The normalized form of a complex fact data type declaration.

§ 5–3 Normalizing Rules

This section discusses a method of normalizing PROLOG and TPROLOG program rules. The method is similar to Ramakrishnan's method of transforming Horn clauses form into canonical form [Ramakrishnan 1987]. Ramakrishnan's method takes a set of Horn clauses (i.e. a PROLOG program) and produces another set of clauses in which all

their arguments are variables and all occurrences of a function symbol (in the Horn clauses) are replaced by a unique occurrence of an infinite relation. Therefore, the canonical form for Horn clauses is a PROLOG program which is free of non–variable terms. For example, the canonical form of rules given in figure 1–2 is as follows:

```
glaswegian_infant (LN, FN, Age) :-person(FN, LN,A,Age),
                                 raddress(_, B, A), b(B),
                                 e(E),
                                 Age < E.

glaswegian_emp( Ln, Fn,  Sch,Yr) :- emp(Fn, Ln, _, C), h(_,_, D,C),
                                 rdegree2(_, _, Sch, Yr, D),
                                  person(Fn, Ln, A,_),
                                 raddress(_, B, A), b(B),
                                 Yr > 1960,
                                 Yr < 1990.
h(_,_, D, C).
rdegree2(_, _, Sch, Yr, D).
raddress(_, B, A).
e(4).
b(glasgow).
```

Hsiang's approach [Hsiang 1985], is similar to Ramakrishnan's method. However, the difference between those two approaches is that whilst Ramakrishnan's method generates unit clauses and stores them as relational facts, Hsiang's approach does not. Thus, the disadvantage of Hsiang's approach is that the new predicates may be not be logical consequences of the program.

Although our method is similar to the above two approaches in general, there are some differences. The differences are as follows:

a) Both approaches assume that all variables are of simple type, whilst our approach considers structured types.

b) Ramakrishnan's method allows a new predicate to be stored as infinite relation, whilst our approach allows finite relations only (i.e. no unit clause is allowed).

c) Ramakrishnan's method replaces every constant by a new finite relation, whilst our approach does not replace constants.

This section divides into two sub–sections. The first sub–section discusses a body predicate normalization and the second sub–section discusses the rule head normalization.

§ 5–3–1 Normalizing Body Predicates

A body predicate is either an arithmetic predicate or a base predicate: a predicate which may unify with a fact or a rule head. We assume that operands in arithmetic predicates are of numerical type and we do not consider them further. A term v in a base body predicate is either a constant, a structured term (i.e. complex term or list), or a variable.

a) If v is a constant, then there is no need for further normalization.

b) If v is a structured term, then the normalization of the base body predicate is similar to fact normalization (q.v. § 5–2–2). However, instead of replacing v by a constant, it is replaced by a variable. Moreover, the generated predicates are added to the body predicates. So, if we have the following base body predicate.

$:- p(x_1, \ldots, x_{k-1}, f1(xf1_1, \ldots, xf1_m), x_{k+1}, \ldots, x_n).$

Then, we replace it by

$:- p(x_1, \ldots, x_{k-1}, V, x_{k+1}, \ldots, x_n),\ rf1(xf1_1, \ldots, xf1_m, V).$

where V is a new variable and $rf1(xf1_1, \ldots, xf1_m, V)$ is a new body predicate

Note that the above example contains one complex term, we can easily generalize the method to several structured terms.

c) If v is a variable of structured data type t, then

1) If t is a list data type or variant type, then a new unary complex term is generated as shown in § 5–2–1–3. Similarly, v is replaced by the unary complex term. After that, the body predicate is normalized as shown in (b) above and a generated body predicate is normalized as shown in (3) below. For example, for a given body predicate emp(Fn, Ln, D), where D is a variable of list1([{f2, f3}]) (q.v. figure 3–1), D is replaced by a unary complex term list1(D), where D is of variant type [f1, f2], and then emp(Fn, Ln, list1 (D)) is replaced by emp(Fn, Ln, V), rlist1(D, V). Further normalization on rlist1(D, V) is shown in (3) below.

2) If t is a complex term type $f1(af_1: tf_1, \ldots, af_m: tf1_m)$ (q.v. chapter 3), then we replace v by a new complex term $f1(xf1_1, \ldots, xf1_m)$ (where $(\forall\ 1 \le i \le m)\ xf1_i$ is a variable of type $tf1_i$). The normalization of the generated complex term are explained in (b). Note, the replaced variable in normalizing processor is v. For example, the body predicate emp(Fn, Ln, [_, _, degree2(_, _, Sch, Yr)]) in figure 1–2

contains Sch variable of complex term type

f4(school(name : school_name, school_city : city), then Sch is replaced by school(_, _) term and then the body predicate is normalized as shown in figure 5–3.

3) If t is a variant type, then the normalization of v is similar to § 5–2–2 (c). However, there are two differences. Firstly, instead of replacing the variant type by attribute name is replaced by v. Secondly, all new predicates are disjointed together and added to the body predicate. For example, rlist1(D, V) in (1) above is replaced by rlist1(D, V), (rdegree1(_, _, D); rdegree2(_, _, _, _, D)).

As an example, the normalization of body predicate rules given in the program of figure 1–2 is in figure 5–3.

```
:-person(FN, LN, V2,_), raddress(V3, glasgow, V2),
        (rnone(V3); rflat(_,_,_,V3); rhouse(_,_,V3)).
:- emp(Fn, Ln, _, V1),
        rlist1(V2,V1), (rdegree1(_, _, V2);(rdegree2(_,_,V3,_,V2), rschool(_, _, V3))),
        rlist1(V4, V1), (rdegree1(_, _, V4); (rdegree2(_,_,V5,_,V4), rschool(_, _, V5)),
        rlist1(V6, V1), rdegree2(_, _, Sch, Yr, V6), rschool(_, _, Sch), person(Fn, Ln, V7,_),
        raddress(V8, glasgow, V7), (rnone(V8); rflat(_,_,_,V8); rhouse(_,_,V8)),
        Yr > 1960, Yr < 1990.
```

Figure 5–3. the normalized rule body predicates of figure 1–2.

§ 5–3–2 Normalizing Rule Heads

§ 5–3–1 describes the normalization of the body predicates. This section describes the normalization process of a rule head. Note, we assume that

body predicates are normalized by the method in § 5–3–1.

A term v in a rule head is either a constant, a variable, a complex term, or a list.

a) If v is a constant or a variable, then there is no need for further normalization.

b) If v is a complex term, then the removal and replacement of v is defined as follows: Given the following rule:

$p(x_1, \ldots, x_{k-1}, f1(xf1_1, \ldots, xf1_m), x_{k+1}, \ldots, x_n) :- p_1, \ldots, p_g.$

Then, we replace the rule by

$r(x_1, \ldots, x_{k-1}, V, x_{k+1}, \ldots, x_n) :- rf1(xf1_1, \ldots, xf1_m, V), p'_1, \ldots, p'_g.$

where V is a new variable, $p'_1, \ldots, p'_g$ are the normalized body predicates $p_1, \ldots, p_g$ and $rf1(xf1_1, \ldots, xf1_m, V)$ is a new body predicate. Note, the new body predicate is valid, because it will be asserted as a temporary fact at execution time. If p'_k $(1 \leq k \leq g) = rf1(xf1_1, \ldots, xf1_m, V)$, then there is no need to duplicate it.

c) If v is a list, then v is represented either by a list of terms or as a head and tail.

1) If v is a list consisting of n terms, then v is transformed into a unary complex term instead of v (q.v. § 5–2–1–2 (1)). The transformed rule head is normalized in the same way as in (b) above. Further normalization for the generated body predicate is

needed (q.v. § 5–3–1 (b)).

2) If v consists of a head X and tail Y, then the rule is transformed into two rules (q.v. (1) above). The normalization of the two rules is similar to (1) above. Note, Y is variable of the list type which needs further normalization as a body predicate (q.v. § 5–3–1).

The following is an example of normalizing rules; if we have rules given in a program of figure 1–2, then the normalization of it is as follows:

```
glaswegian_infant(LN, FN, Age):-person(FN, LN, V2,_),
        address(V3, glasgow, V2),
        (rnone(V3); rflat(_,_,_V3); rhouse(_,_V3)).
glaswegian_emp(Ln,Fn, Sch, Yr):-
        emp(Fn, Ln, _, V1),
        rlist1(V2,V1), (rdegree1(_, _, V2);(rdegree2(_,_V3,_,V2), rschool(_, _, V3))),
        rlist1(V4, V1), (rdegree1(_, _, V4); (rdegree2(_,_,V5,_,V4), rschool(_, _, V5)),
        rlist1(V6, V1), rdegree2(_, _, Sch, Yr, V6), rschool(_, _, Sch), person(Fn, Ln, V7,_),
        raddress(V8, glasgow, V7), (rnone(V8); rflat(_,_,_V8); rhouse(_,_V8)),
        Yr > 1960, Yr < 1990.
```

Figure 5–4. The normalized form of rules in figure 1–2.

§ 5–4 Goal Normalization

A goal G is correctly answered (i.e. G is a theorem) on a program P, if it is a logical consequence of P. Theorem proving for PROLOG is based on SLD–refutation via depth–first and left–most computation rule. Therefore, in order to answer a goal, the left–most sub–goal of the goal

should be unified with either a fact or a rule head in the program, and the so on for the rest of sub–goals.

Hence, in order to normalize a goal, we have to normalize it in the same way as its unified atom (i.e. a fact or a rule head).

a) If a sub goal is unified with a fact, then

1) Each a structured term in the sub–goal should be normalized in the same way as in § 5–2. However, the only difference is that instead of replacing a structured term by a constant, it is replaced it by a new variable.

2) Each variable is normalized in the same way as in § 5–3–1 (c).

b) If a sub–goal is unified with a rule head, then

1) Each structured term is normalized in the same way as in § 5–3–2. However, the only differences are as follows: instead of replacing the structured term by a variable it is replaced by a constant, and instead of having body predicates they become temporary facts in the program(if they are not already exist).

2) If a term is a variable which is unified with a structured term, then the variable is replaced by the unified structured term. The latter is normalized in the same way as in (1).

For example, if we have the following goal ?- glaswegian_emp(L, F, S, Y), then it unifies with the rule glaswegian_emp in figure 1–2, and it is replaced by the following:

```
?- glaswegian_emp(L, F, S, Y), rschool(_, _, S).
```

§ 5–5 Discussion

It is interesting to compare our approach with others. The main difference is that; while [Zaniolo 1985] introduces ERA to make database systems applicable for logic programs, we normalize logic programs to make them applicable to conventional relational database systems. We solved the problem of allowing infinite relations to be generated in the canonical form for Horn clauses [Ramakrishnan 1987] approach, by asserting temporary facts at execution time. Since the canonical form for Horn clauses replaces each constant by a new predicate, a huge amount of facts and rules are generated which may not be used. Temporary facts, in our approach, are generated when they are needed. Our approach allows us to get the [Ramakrishnan 1987, Hsiang 1985] benefits and avoiding their disadvantages.

Our data schema for PROLOG (i.e. TPROLOG) allows variant types, whilst database systems do not. We can integrate database systems and TPROLOG by assuming any argument of variant type is a complex argument, and then we remove the complex argument and replace it by skolem constant or variable as explained in § 5–2 and § 5–3.

Chapter 6: System Architecture

§ 6–1 Introduction

§ 1–2 gives a general description of a system which is proposed. § 3–3 introduces TPROLOG, whilst chapter 4 discusses the safety checking of TPROLOG. Chapter 5 describes the normalization method of normalizing TPROLOG programs.

This chapter describes a design of a compiler which compiles a TPROLOG program into relational algebra expressions. The compiler consists of three translators which perform the following transformations:

1) The translation of a TPROLOG program into a standard PROLOG program (i.e. C–PROLOG program). It is described in § 6–2.

2) The translation of a standard PROLOG program into a PROLOG program which is free of complex arguments. Hereafter, it is referred as a complex–free program. The translation process for PROLOG programs and goals is explained in § 6–3. We have to note that, in order to complete the translation, there is a need for type checking (q.v. § 3–3) and safety checking (q.v. chapter 4) of the program and query.

3) The translation of a complex–free program into RAEs. This is discussed in § 6–4.

The configuration of the system is shown in figure 6–1, and figure 6–2.

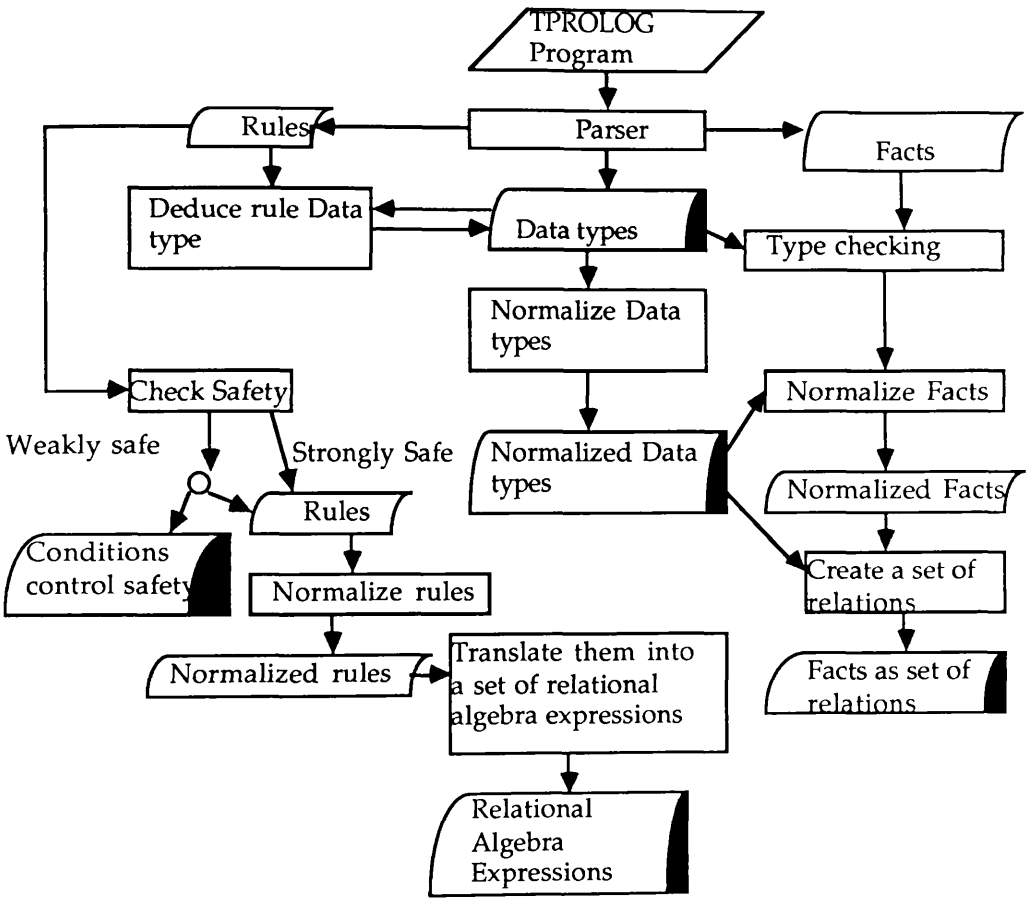


Figure 6–1. System Architecture for Compiling TPROLOG programs into relational algebra expressions.

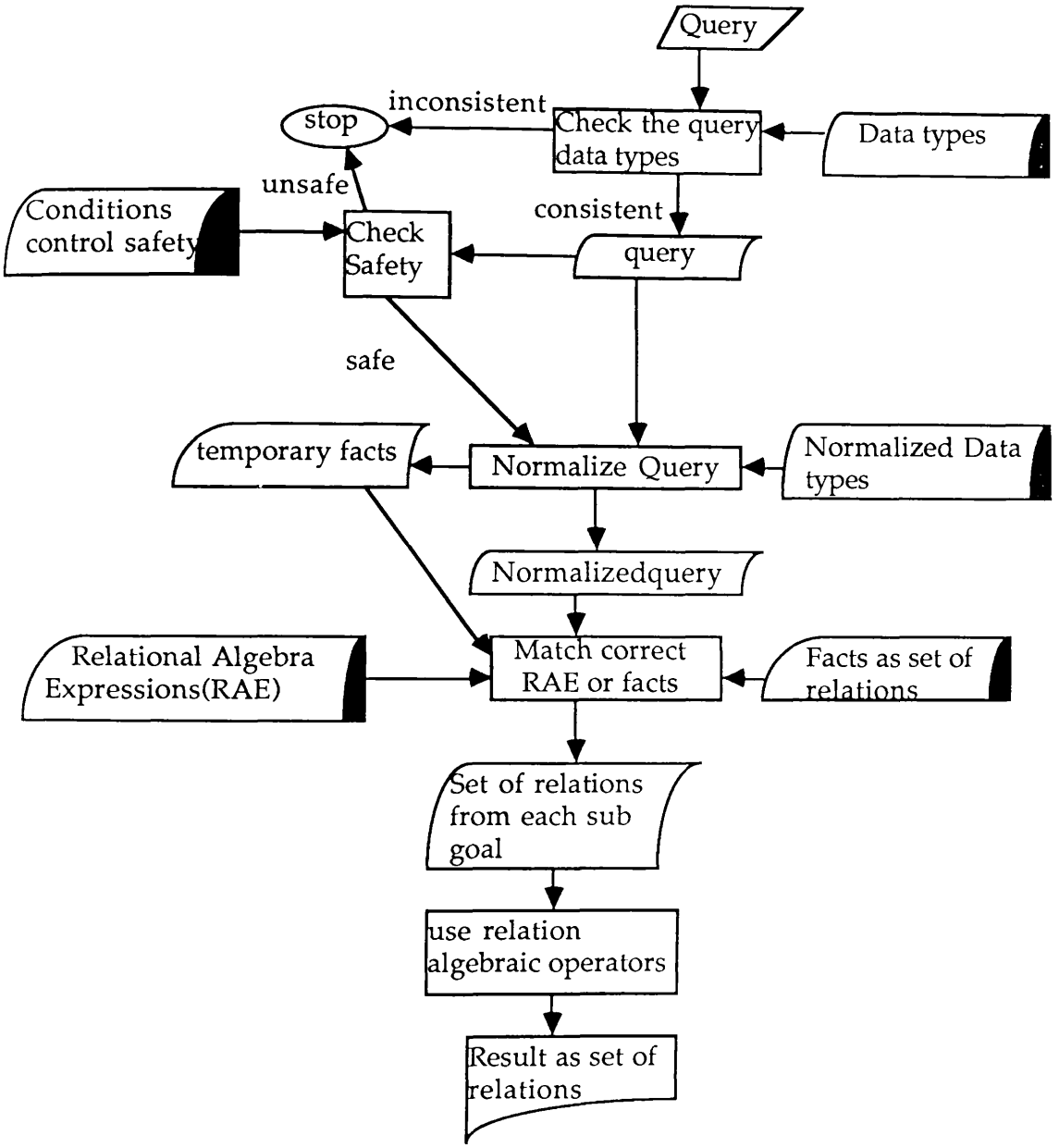


Figure 6–2. System Architecture for Compiling TPROLOG query into relational algebra expressions.

§ 6–2 Translation of TPROLOG into PROLOG

§ 3–3 Introduces TPROLOG. It is, simply, an extension of a standard PROLOG (e.g. C–PROLOG) which is a strongly typed language. Our aim is to transform TPROLOG programs into complex–free PROLOG programs, and then to use one of the existing approaches to transform complex–free PROLOG programs to relational algebraic expressions. Therefore, in order to transform a TPROLOG program into a complex–free PROLOG program we use the following procedure: Parser, Type_checking, and Deduce_rule_data_type in figure 6–1 and check the query data type, and check the query syntax in figure 6–2.

§ 6–2–1 Parser

Parser : TPROLOG program →
Bool × set(rule) × set(fact) × set(fact–declaration) × set(data–type)†

Parser is a procedure which translates a TPROLOG program into four data sets written in a standard PROLOG programming language form. It takes as input a TPROLOG program and checks its syntax with respect to the TPROLOG EBNF (q.v. Appendix A). If it is syntactically valid, then the parser translates it into four data sets.

The set(fact–declaration) and set(data–type) define the TPROLOG program data types. The set(rule) and set(fact) contain the TPROLOG program rules and facts respectively (i.e a PROLOG program).

† → is a function space constructor
× is product constructor

Note that TPROLOG queries are identical to PROLOG queries. Therefore, there is no need to translate it into PROLOG form.

§ 6–2–2 Type Checking

Type_checking : set(rule) × set(fact) × set(fact–declaration) × set(data–type)
→ Bool
Type_checking : set(goals) × set(fact–declaration) × set(data–type)
→ Bool

§ 6–2–1–1 checks the syntax of a TPROLOG program. PROLOG is not strongly typed language, whilst TPROLOG is. Therefore, before a TPROLOG program is translated, it should be type checked.

Type_checking in figure 6–1 and figure 6–2 is a procedure used to check correctness of facts data type and the consistency of goals and body predicates with respect to its type definitions. It is fully explained in chapter 3.

§ 6–2–3 Deducing Data type forRules

Deduce_rule_data_declarations : set(rule) × set(data–declaration) × set(data–type)
→ Bool × set(data–declaration)

The Deduce–rule_data_declaration is a procedure used to deduce the data type of rule heads. It takes as an input the type information (i.e. the type information of facts (q.v. § 6–2–1) and the deduced rule data type). Before the rule head data type is deduced, the body predicates are typed and checked for consistency.

§ 6–3 Translation of TPROLOG Programs into Complex–Free PROLOG Programs

§ 6–2 describes the procedure which translates a TPROLOG program into an equivalent PROLOG program. This section describes the first step of compiling a PROLOG program into equivalent RAEs: normalization.

In chapter 2 we showed that a safe PROLOG program is equivalent to RAEs. Therefore, before the compilation of a PROLOG program is carried out, the PROLOG program is safety checked (q.v. chapter 4) and normalized (q.v. chapter 5), in that order.

The description of the safety checking procedure is in § 6–3–1. § 6–3–2, § 6–3–3, and § 6–3–4 describe the normalization of a PROLOG program, whilst § 6–3–5 describes the goal normalization.

§ 6–3–1 Safety Checking

§ 6–3–1–1 Safety Checking for Rules

Check_rules_safety : (set(rule) × set(facts) → rule/goal graph)
→ Bool × set(safe–path)

The Check_rules_safety, which check the safety of a PROLOG program, is a decision procedure which takes as input a rule–set a and fact–set. The procedure generates a graph which represents the execution route of the rule–set (i.e. rule/goal graph), and then checks the safety of the rule–set graph in the light of safety conditions. The result of the decision is either the rule–set is strongly safe or it is weakly safe (q.v. chapter 4). Note that, the (weakly and strongly) safe paths represent all safe executions of the rules.

§ 6–3–1–2 Safety Checking for Goals

Check_goals_safety : set(goals) × set(safe–path) → Bool

Check_goals_safety is a decision procedure which takes a set of sub–goals and safe execution paths as an input. The procedure checks the safety of each sub–goal with respect to the safe execution paths. This is done by generating a goal graph and mapping it into the safe execution paths. The result of the decision depends on whether the goal graph is successfully unified with a safe execution path or not.

§ 6–3–2 Normalizing Data Declarations

Normalize_data_declaration : set(data–declaration)× set(data–type)
→ set(normalized–declaration)

Normalize_data_declaration is a procedure which takes the type information of a PROLOG program (i.e. data_declaration_set and data_type (q.v. § 6–2)). It is used to extract and replace structured type terms by simple type terms. It outputs a new set of type information which is equivalent to the original type information. The new set of type information contains a simple type terms only.

§ 6–3–3 Normalizing Facts

Normalize_facts :
set(data–type) × set(facts) × set(normalized–declaration) ×
set(facts–declaration) → Bool × set(normalized–fact)

Normalize_facts is a procedure which extracts structured terms and replaces them by new simple terms in the same way as their respective types are normalized. It takes as an input the set of facts in the program,

the type information about the facts, and the normalized form of the type information. Before the facts are normalized, they are typed checked (q.v. § 6–2–2). If the ground terms in facts are correctly typed, the normalize procedure is carried out. If a fact is ill typed, then it is rejected and will not be normalized. The output of the normalize procedure is a new set of facts which are equivalent to the original facts (q.v. § 5–2–1). The new facts are complex–free facts.

§ 6–3–4 Normalizing Rules

Normalize_rules : set(data–type) × set(normalized–declaration) × set(rule)
→ set(normalized–rule)

Normalize_rules is a procedure used to extract structured terms from rule heads and their body predicates by replacing them by new simple terms. It takes a PROLOG program rule and produces an equivalent new set of rules which does not contain any structured terms. The procedure is carried out, after the typing checking done on its body predicates and the rule head types are deduced, and safety of the program are done. It produces a new set of complex–free rules which are equivalent to the original rules.

§ 6–3–5 Normalizing Goals

Normalize_goals :
set(goal) × set(data–type) × set(normalized–declaration) ×
set(data–declaration) → set(normalized–query) × set(temporary–fact)

Normalize_goals is a procedure which extracts all complex terms from goals and replaces them by new simple terms. It is carried out, after the goals are correctly typed and safety checked. It takes as input a set of PROLOG goals, the original type information, and the normalized form of the type informations. It outputs a new–goals–set which is a normalized

form of the goals. The normalization process for goals is similar to the normalization of the unified clause. The only difference is some temporary facts may be added to the program.

§ 6–4 The Transformation of Complex–free PROLOG programs into Relational Algebraic Expressions (RAEs)

§ 6–3 describes how a TPROLOG program is transformed into a complex–free PROLOG program. This section shows how a relational database can be constructed from a complex–free PROLOG program. The translation of a complex–free PROLOG program is based on Reiter's, Henschen's, Chang's and Bancilhon's approaches [Reiter 1978, Hencschen 1984, Chang 1986, Bancilhon 1986]. The construction is done as follows:

a) The fact base (i.e. normalized–facts and their type information) is transformed into a base–table. Each base–table is represented in storage by a distinct stored file.

b) The rules base (i.e. PROLOG complex–free rules and their type information) is translated into a view. A view is a table which does not exist in its own right, but instead it is derived from one or more tables (i.e. view or base–table).

Chapter 7: Implementation

§ 7–1 Introduction

Chapter 6 discusses the system architecture. This chapter discusses the implementation of the system using C–PROLOG.

The system is divided into two separate parts. One for compiling a TPROLOG program into RAEs (figure 6–1) and the other for compiling a TPROLOG query into a relational algebraic query languages (figure 6–2).

This chapter consists of three sections. § 7–2 discusses the implementation of the TPROLOG compiler, whilst § 7–3 discusses the TPROLOG query compiler. § 7–4 gives the status of the implementation.

Note that, terminals in Appendix A are referred to in this chapter. Moreover, they are written in italic form.

§ 7–2 The Compilation of TPROLOG Programs

§ 7–2–1 The Transformation of TPROLOG Programs into Complex–Free PROLOG Programs

§ 6–2 and 6–3 describe the transformation of TPROLOG programs into complex–free PROLOG programs. The transformation is done in two steps; the transformation of a TPROLOG program into a PROLOG program, and then the transformation of a PROLOG program into a complex–free PROLOG program. This section describes the

implementation of these two steps.

§ 7–2–1–1 The Transformation of TPROLOG Programs into PROLOG Programs

The transformation of a TPROLOG program into a PROLOG program consists of two procedures (q.v. § 6–2): the parser, and the deduction of rule types.

§ 7–2–1–1–1 Parser

parser(P) = (Ok, rule–set, fact–set, facts–declaration–set, data–type–set)

where Ok is false if a TPROLOG program P is not syntactically valid, otherwise Ok is true. If Ok is true, then P is translated into four data set written in PROLOG form..

The output of the parser is as follows:

a) Each *statement* (q.v. Appendix A) of the form *clause*, where *clause* is of the form *structure :- expressions.* (i.e. rule), is translated into an equivalent unit clause called rule' defined as follows:

rule(*structure, expressions*).

where structure is defined as follows:

predicate × list(*term*)

where *predicate* is the rule head predicate name, list(*term*) are the sub–terms in the rule head, and *expressions* is defined as follows:

list(Ψ)× list(list(*term*))

where Ψ is either a body predicate symbol, an *operator* of the arithmetic expression in the body of the rule, a *logicop* in the body of the rule, or *!*, and list(*term*) is list of terms associated with each Ψ. Moreover, rule' ∈ rule–set. For example, the rules in the program of figure 1–2 are translated into the rule–set shown in figure 7–1.

```
rule(glaswegian–infant, [LN, FN, Age], [person, < ],
    [[FN, LN, address(_, ba, glasgow), Age], [Age, 4]]).
rule(glaswegian–emp, [Ln, Fn, Sch, Yr], [emp, person, >, <],
    [[Fn, Ln, _, [_, _, degree2(_, _, Sch, Yr)]], [Fn, Ln, address(_, glaswegian), _],
    [Yr, 1960], [Yr, 1990]]).
```

Figure 7–1. the translated rules in figure 1–2.

b) A set of *statement* of the form *clause*, where each clause is of the form *structure.* (i.e. facts), is translated into an equivalent unit clause called fact', and it is defined as follows:

relation(*predicate*, set(list(*term*)))

where *predicate* is a predicate symbol (i.e. fact name) and each list(*term*) is a list of ground terms for each fact with the same predicate symbol. Moreover, fact' ∈ fact–set. For example, the facts in the program of figure 1–2 are translated as shown in figure 7–2.

```
relation(person, [[joe, cool, porter, address(none, glasgow), 20],
        [max, fax, guard, address(flat(21, 18, windsor_street, g20), glasgow), 40],
        [joe, doe, address(house(31, kew_drive, g12), glasgow), 3]]).
relation(emp, [[joe, cool, porter, none], [max, fax, guard, [degree1(hs,1968)]],
        [fred, red, staff, [degree1(hs, 1975),
        degree2(msc, ba, school(glasgow_university, glasgow), 1980),
        degree2(msc, ba, school(glasgow_university, glasgow), 1983)]]]).
```

Figure 7–2. The rewritten form of facts in figure 1–2.

c) Each *statement* of the form % *facts-declaration.*, with respect to the set of *statement* of form $ *data-type.*, is translated into a set of unit clauses. Such a unit clause, called fact-declaration', is defined as follows:

schem(*predicate*, list(*term*), list(*type*), list(complex-type))

where *predicate* is a fact predicate name or a function symbol in the *data-type* (q.v. d), *term* is an attribute (i.e. *con*) in the *fact-declaration* or *data-type*, and each *term* is associated with a type. If the *data-type* of an attribute in a *fact-declaration* is of variant type, then the predicate definition consists of more than one fact-declaration' for the *fact-declaration*. The list(complex-type) $\subseteq$ list(*type*) is used when we search for an attribute of complex type. Finally, fact-declaration' $\in$ fact-declaration-set. For example, from the type information in figure 3–1 and figure 3–2 of the program in figure 1–2, we get the set of facts shown in figure 7–3.

```
schem(emp, [first_name, last_name, job_name, degree], [name, name, job, f1], [f1]).
schem(emp, [first_name, last_name, job_name,degree], [name, name, job, f2], [f2]).
schem(emp, [first_name, last_name, job_name,degree], [name, name, job, f3], f3]).
schem(person, [first_name, last_name, home_address, person_age],
    [name, name, addresses, age], [addresses]).
```

figure 7–3. The translation of *fact–declaration* of the program in figure 1–2.

d) Each *statement* of the form *$ data–type.* is translated into data–type', where data–type' ∈ data–type–set, and data–type' is defined as follows:

1) If *data–type* is of the form *type(<con, integer, integer>)*, then data_type' is of the form type(X) :-*con*(X), between(X, *integer, integer*), where X is a variable. Note, between is a built–in predicate used to check the range of X. It is explained in § 3–3–1.

2) If *data–type* is of the form *type({atom, ..., atom})*, then data_type' is of the form *type*(X) :- member(X, [*atom, ..., atom*]).

3) If *data–type* is of the form *type(predicate)*, then data–type' has the same form as *data–type*.

4) If *data–type* is of the form *type([type*$_1$*, ..., type*$_n$*])*, then data_type' is of the form *type*(X) :- element_in(X,Y), member(Y,[*type*$_1$*, ..., type*$_n$]). Note, element_in is a built_in predicate used to assume that X is of type Y.

5) If *data–type* is of the form *type([{type*$_1$*, ..., type*$_n$*}])*, then data_type'

is of the form *type*(X) :- each_element(X,Z), element_in(Z,Y), member(Y,[$type_1$, ..., $type_n$]). Note, each_element is a built_in predicate used to take an element Z from list X.

6) If data–type is of the form *type(predicate(*con_1: : $type_1$, ..., con_n: $type_n$*))*, then data–type' is of the form *type(predicate(*$type_1$, ..., $type_n$*))*.

An example of translating set of *$ data–type* in figure 3–1 into data–type–set is shown in figure 7–4.

```
name(X):-string(X), between(X, 1, 10).
street(X):- string(X), between(X, 1, 30).
city(X) :- member(X, [glasgow, london, edinburgh, manchester, birmingham, reading]).
undergrad(X) :- member(X, [hs, primary]).
postgrad(X) :- member(X, [msc, phd, diploma]).
subject(X) :- member(X, [ba, computer, engl, math, engineering, biology, medicine]).
school_name(X):- member(X, [glasgow_university, edinburgh_university,
    heriot_watt_university]).
job(X) :- member(X, [ porter, guard, vp, staff]).
post_code(X) :- member(X, [g1, g2, g3, g12, g20]).
age(X) :- integer(X), between(X, 0, 200).
year(X):-integer(X), between(X, 1800, 2100).
house_no(X) :- integer(X), between(X, 1, 1000).
f1(none).
f2(degree1(undergrad, year)).
f3(degree2(postgrad, subject, school, year)).
f4(school (school_name, city)).
addresses(address(home, city)).
list1(X) :- each_element(X, Z), element_in(Z, Y), member( Y, [f2,f3]).
qualification(X) :- element_in (X,Y), member(Y, [f1, list1]).
```

Figure 7–4 The translation of the program in figure 3–1 into data–type–set.

For the sake of type checking, each term of the form *predicate* is translated into schem(*predicate*,[],[], []) , and each term of the form *predicate*'(' *con* ':' *type* {',' *con* ':' *type*} ')' is translated into schem(*predicate*,list(*con*) , list(*type*), list(*complex-type*)). Translated terms are added to fact-declaration-set. For example, the translation of the *$ data-type* in figure 3–1 are translated as shown in figure 7–5.

```
schem(none, [], [], []).
schem(degree1,[degree_name, degree_year], [under_grad, year], []).
schem(degree2, [degree_name, degree_subject, degree_school, degree_year],
    [post_grad, subject, f4, year], [f4]).
schem( school, [name, school_city], [school_name, city], []).
schem(address, [house_address, city_address], [f1, city], [f1]).
schem(address, [house_address, city_address], [house_address, city], [house_address]).
schem(address, [house_address, city_address], [flat_address, city], [flat_address]).
schem(flat, [flat_no, building, street_name, code], [integer, house_no, street, post_code],
    []).
schem(house, [building, street_name, code], [house_no, street, post_code], []).
```

Figure 7–4 The translation of *$ data-type* set in figure 3–1 into a fact-declaration-set

§ 7–2–1–1–2 Deducing Rule Data Types

```
Deduce_rule_data_type(rule-set, data-declaration-set, data-type, ) =
                    ( Ok,rule-data-declaration-set)
```

Deduce_rules_data_type is a recursive procedure used to produce a rule-declaration-set (i.e. type of variables in rule heads), where a

rule–declaration $\in$ rule–declaration–set is syntactically equivalent to the syntax of the fact–declaration–set (q.v. § 7–2–1–1–1 (c)) (note, an attribute name may be *var* in a rule head). The procedure takes as an input data–declaration–set, and a rule–set (q.v. § 7–2–1–1–1), where

data–declaration–set= fact–declaration–set $\cup$ rule–declaration–set

It checks the correctness and the consistency of the body predicates data type with respect to its existing type information (i.e. data–declaration–set and data–type). The output of the procedure is defined as follows: For each R $\in$ rule–set, D $\subseteq$ data–declaration–set , and T $\subseteq$ data–type–set,

a) If the procedure can deduce type for R head from D and T (i.e. OK is true), then rule–declaration is generated and added to D. Note that, if a variable in head of R has variant data–type, then there is more than one rule–declaration for R.

b) Otherwise, R is untyped (i.e. OK is false). As a result of this decision we cannot compile R to a RAE. Therefore, there is no need for further processing.

For example, from figure 7–3, figure 7–4, and figure 7–5 the deduced rules data type of the program in figure 1–2 is shown in figure 7–6.

```
schem(glaswegian_emp, [Ln, Fn, Sch, Yr], [name, name, school, year], [school]).
schem(glaswegian_infant,[LN, FN, Age], [name, name, age], []).
```

Figure 7–6. A rules data declaration of the program in figure 1–2.

§ 7–2–2 The Transformation of PROLOG Programs into Complex–Free PROLOG Programs

§ 7–2–1 showed that further processing should be done only on facts and rules which have a type. Therefore, the transformation of a TPROLOG program into a complex–free PROLOG program procedure assumes facts and rules which are correctly typed.

§ 7–2–2–1 Normalizing Data Declarations

Normalize_data_declaration (data–declaration, data–type) =
(normalized–declaration).

Each normalized–declaration ∈ normalized–declaration–set (q.v. § 6–3–2) is produced by replacing each term (i.e. constant or variable) of a complex data type in each data–declaration ∈ data–declaration–set with a new term (i.e. constant or variable) of skolem constant type. Moreover, a complex type is introduced as normalized–declaration by adding the replaced term and its type to it, and prefixing r to the function name (q.v. § 5–2). Note, the data declaration of complex terms is introduced in § 7–2–1–1–1. A normalized–declaration is of the following form:

new_schem(*predicate*,list(*term*) × list(*type* + skolem–type))

For example, the normalized form of data–declaration–set, which is shown in figure 7–3, figure 7–5, and figure 7–6, is shown in figure 7–7.

```
new-schem( emp, [first_name, last_name, job_name, a1], [name, name, job, skolem]).
new-schem( person, [first_name, last_name, a3, person_age], [name, name, skolem, age]).
new_schem(rnone, [a1], [skolem]).
new_schem(rdegree1, [degree_name, degree_year, a4], [under_grad, year, skolem]).
new_schem(rdegree2,[degree_name, degree_subject, a2, degree_year, a4],
    [post_grad, subject, skolem, year, skolem]).
new_schem(rschool,[name, school_city, a2], [school_name, city, skolem]).
new_schem(rlist1, [a4, a1], [skolem, skolem]).
new_schem(raddress, [a1, city_address, a3], [skolem, city, skolem]).
new_schem(rflat, [flat_no, building, street_name, code, a1],
    [integer, house_no, street, post_code, skolem]).
new_schem(rhouse, [building, street_name, code, a1],
    [house_no, street, post_code, skolem]).
new_schem(glaswegian_emp, [Ln, Fn, Sch, Yr], [name, name, skolem, year]).
new_schem(glaswegian_infant, [LN, FN, Age], [name, name, age]).
```

Figure 7–7. The normalization of data set in figure 7–3, figure 7–5. and figure 7–6.

§ 7–2–2–2 Normalizing Fact Base

§ 7–2–2–2–1 Facts Type Checking

facts_type_checking(fact-set, facts-declaration-set, data-type-set) = Ok

The type of each fact in fact-set is checked with respect to its fact declarations in facts-declaration-set and data-type-set (q.v. § 3–3–2). The output of the type checking result is either typed (i.e. Ok is true) or untyped (i.e. Ok is false). If Ok is false , then the fact cannot be stored in a database. Therefore, there is no need for further processing. An example of type checking is that relation emp and person in figure 7–2 are typed with respect to data-declaration-set in figure 7–3 and figure 7–5 and data-type-set

in figure 7–4.

§ 7–2–2–2–2 Normalizing Facts

Normalize_fact (data–type, fact–set, fact–declaration–set, normalized–declaration–set)=
(Ok, normalized–fact–set).

normalize_fact procedure checks the type of a ground term in the fact–set (q.v. § 7–2–2–2–1) . If the ground terms data type is correct (i.e. Ok is true), then the procedure outputs a normalized–facts–set (q.v. 5–2–1). A normalized–facts–set is a rewritten form of the fact–set (q.v. § 7–2–1). In general, the procedure mirrors the facts–declaration normalization (q.v. § 7–2–2–1). More precisely, the transformation from fact–set to normalized–facts–set is done recursively as follows:

a) Each complex term in a fact' ∈ facts–set is replaced by a new skolem constant.

b) Each complex term is transformed into a new fact called normalized–fact, where normalize–fact ∈ normalized–facts–set, by appending the complex term with the replaced skolem constant.

A normalize–fact ∈ normalized–fact–set is of the following form:

new–relation(*predicate*, set(list(*con*)).

where *predicate* is either fact predicate name or a functor of complex term in a fact, and *con* is is either a constant in a fact–set or a constant of skolem type. For example, relation emp and person in figure 7–2 are typed with respect to data–declaration–set in figure 7–3 and figure 7–5 and data–type–set in figure 7–4. Therefore, the normalized form of facts emp and person is

shown in figure 7–8 with respect to normalized–data–declaration in figure 7–7.

```
new–relation(person, [[joe, cool, c1, 20], [max, fax c3, 40], [joe, doe, c5, 3]]).
new–relation(emp, [[joe, cool, porter, c7], [max, fax, guard, c8], [fred, red, staff, c9]]).
new_relation(raddress, [[c2, glasgow, c1], [c4, glasgow, c3], [c6, glasgow, c8]]).
new_relation(rnone, [[c2], [c7]]).
new_relation(rflat, [[21, 18, windsor_street, g20, c4]]).
new_relation(rhouse, [[31, kew_drive, g12, c6]]).
new_relation(rlist1, [ [c14, c8], [c11, c9], [c12, c9], [c13, c9]]).
new_relation(rdegree1, [[hs, 1968, c14], [hs, 1975, c11]]).
new_relation(rdegree2, [ [msc, ba, c10, 1980, c12], [phd, ba, c10, 1983, c13]]).
new_relation(rschool, [[glasgow_university, glasgow, c10]]).
```

Figure 7–8. The normalized form of facts in figure 7–2.

§ 7–2–2–3 Normalizing Rule Base

§ 7–2–2–3–1 Rules Safety Checking

```
Check_rules_safety ( rule/goal graph generator( rule–set)) =
                    (Ok, safe–path–set)
```

check_rules_safety procedure works in two steps.

step 1 : Takes as an input rule–set and generates a graph which represents the execution path of the rule–set (q.v.§ 4–3). Each node in the graph is represented in the following form.

```
node( current_node,
    list_of_nodes_directed_from_current_node,
    list_of_nodes_directed_to_current_node)
```

step 2 : After the rule/goal graph of rule-set is generated, the procedure checks the safety of the rule-set in the light of safety conditions (q.v. § 4–2–2). The result of the decision may be defined as follows: For each P $\subseteq$ rule-set (where P is a set of rules represented by a graph using the rule/goal graph generator),

a) P is strongly safe (i.e. OK is false), if every execution path for P in the rule/goal graph satisfies the safety conditions. In this case, there is no need to store the graph.

b) Otherwise, we say P is weakly safe (i.e. Ok is true). In this case, some execution paths for P satisfy the safety conditions, but not all. The sub-graph of the P which satisfies the safety conditions is called safe-path-set. Note, if there is no path is satisfied by the safety conditions, then safe-path-set is empty.

For example, each execution path for each rule in figure 7–1 satisfies the safety conditions: whole program is strongly safe and there is no need to store the graph.

§ 7–2–2–3–2 Normalizing Rules

Normalize_rules(rules-set, data-declaration, normalized-declaration-set) =
normalized-rule-set

The procedure takes rule-set (q.v. § 7–2–1–1–1 (a)), data-declaration (q.v. § 7–2–1–1–1 (c)), and normalized-declaration-set (q.v. § 7–2–2–1) and produces a new-rules-set. The new-rules-set is produced by extracting complex terms from each rule in rules-set. The extraction is defined as follows: For each R $\in$ rules-set

a) Each complex term in R head is replaced by a variable of skolem constant type. Each complex term is added to the body predicates of R, after appending the complex term with replaced variable.

b) Each complex term in the body of R is replaced by a new variable of skolem constant type. Each complex term is add to the body of R, after appending the complex term with the replaced variable. Note, if the complex term also exists in R head, then it is replaced by the same variable. Moreover, the type of each variable of a complex term data type is replaced by the skolem constant type.

A normalized-rule $\in$ normalized-rule-set is of the following form

new-rule(normalized-structure, normalized-statement)

where

normalized-structure = *con* $\times$ list(non-structured-term), and
normalized-statement = list($\Psi \times$ list(non-structured-term))

where Ψ is defined in §7-2-1-1-1. For example, rule-set in figure 7-1 are translated into a set of new-rules as shown in Appendix B.

§ 7-2-3 The Transformation of Complex-free PROLOG Programs into RAEs

§ 7-2-1 and § 7-2-2 described how a TPROLOG program is transformed into a complex-free PROLOG program. A complex-free PROLOG program consists of the following:

1) Normalized–data–declaration–set (q.v. § 7–2–2–1).

2) Normalized–facts–set (q.v. § 7–2–2–2–2).

3) Normalized–rule–set (q.v. § 7–2–2–3–2).

This section shows how a relational database can be constructed from the above components. In general, the construction is done as follows:

a) Each normalized–fact ∈ normalized–fact–set is transformed into a base–table. Each base–table is represented in storage by a distinct stored file.

b) Each normalized–rule ∈ normalized–rule–set is translated into a view. A view is a table which does not exist in its own right, but instead it is derived from one or more tables (i.e. view or base–table).

In the following sub–sections we discuss, in more detail, the translation of complex–free PROLOG program into RAE.

§ 7–2–3–1 Storing Facts in Database

Normalized facts and their data declarations are transformed into a set of RAE. Facts are stored in the data base by executing the set of RAEs. The transformation of normalized facts and their declarations into a set of RAEs is defined as follows:

a) For each normalize–fact–declaration ∈ normalized–fact–declaration–set, an empty base–table can be created (i.e. relational schem) using the CREATE–TABLE operation. So that, each normalized–fact–declaration is transformed as follows:

CREATE–TABLE *predicate* (list (*term* × *type*))

where *predicate* is a fact name and *term* is an attribute name, and *type* is a type of the corresponding attribute. Note, we assume that user–define types (i.e. domain) are supported by the relational data base management systems. For example, the normalized–data–set of facts in figure 7–7 is transformed into a set of RAE shown in figure 7–9.

```
CREATE–TABLE emp( first_name : name, last_name : name, job_name : job, a1 : skolem)
CREATE–TABLE person( first_name : name, last_name : name, a3 : skolem,
        person_age : age)
CREATE–TABLE rnone( a1 : skolem)
CREATE–TABLE rlist1 (a1 : skolem, a4 : skolem)
CREATE–TABLE rdegree1( degree_name : under_grad, degree_year : year, a4 : skolem)
CREATE–TABLE rdegree2 ( degree_name : post_grad, degree_subject : subject, a2 : skolem,
        degree_year : year, a4 : skolem)
CREATE–TABLE rschool ( name : school_name, school_city : city, a2 : skolem)
CREATE–TABLE raddress(a1 : skolem, city_address : city, a3 : skolem)
CREATE–TABLE rflat(flat_no : integer, building : house_no, street_name : street,
        code : post_code, a1 : skolem)
CREATE–TABLE rhouse( building : house_no, street_name : street, code : post_code,
        a1 : skolem)
```

Figure 7–9. The transformation of facts data declaration in figure 7–7 into RAE.

b) For each normalized–fact ∈ normalized–fact–set, each list(con) (q.v. § 7–2–2–2–2) is transformed as follows:

INSERT INTO *predicate* (list(*term*)) : list(*con*)

where *predicate* is a fact name and list(*term*) is a list of attributes name (q.v. § 7–2–2–1) corresponds a type of list(*con*) of arguments in the fact. For example, the set of new_relation in figure 7–8 is transformed into RAEs shown in figure 7–10.

```
INSERT INTO person (first_name, last_name, a3, person-age) : joe, cool, c1, 20;
INSERT INTO person (first_name, last_name, a3, person-age) : max, fax, c3, 40;
INSERT INTO person (first_name, last_name, a3, person-age) : joe, doe, c5, 3;
INSERT INTO raddress(a1, city_address, a3) : c2, glasgow, c1;
INSERT INTO raddress(a1, city_address, a3) : c4, glasgow, c3;
INSERT INTO raddress(a1, city_address, a3) : c6, glasgow, c5;
INSERT INTO rhouse(building, street_name, code, a1) : 31, kew_drive, g12, c6;
INSERT INTO rflat(flat-no, building, street_name, code, a1) : 21, 18, windsor_street, g20,
        c4;
INSERT INTO emp (first_name, last_name, job_name, a1) : joe, cool, porter, c7;
INSERT INTO emp (first_name, last_name, job_name, a1) : max, fax, guard, c8;
INSERT INTO emp (first_name, last_name, job_name, a1) : fred, red, staff, c9;
INSERT INTO none (a1) : c2;
INSERT INTO rnone (a1) : c7;
INSERT INTO rlist1 (a4, a1) : c14, c8;
INSERT INTO rlist1 (a4, a1) : c11, c9;
INSERT INTO rlist1 (a4, a1) : c12, c9;
INSERT INTO rlist1 (a4, a1) : c13, c9;
INSERT INTO rdegree1 (degree_name, degree_year, a4) : hs, 1968, c14;
INSERT INTO rdegree1 (degree_name, degree_year, a4) : hs, 1975, c11;
INSERT INTO rdegree2 (degree_name, degree_subject, a2, degree_year, a4) :
        msc, ba, c10, 1980, c12;
INSERT INTO rdegree2 (degree_name, degree_subject, a2, degree_year, a4) :
        phd, ba, c15, 1983, c13;
INSERT INTO rschool (name, school_city, a2) : glasgow_university, glasgow, c10;
INSERT INTO rschool (name, school_city, a2) : glasgow_university, glasgow, c15;
```

Figure 7–10. The transformation of new_relation in figure 7–8 into a set of RAEs.

§ 7–2–3–2 Rules Transformation

A predicate definition, in a PROLOG program, is one of the following:

1) A predicate definition consists of one non–recursive rule.

2) A predicate definition consists of more than one clause (i.e. rules and facts). Note, it may contains recursive rules.

In the following sub–sections we discuss the transformation of each of the above predicate definition into a view. The first sub–section discusses the transformation of predicate consisting of one non–recursive rule, and the other sub–section discusses the generalization of the transformation procedure in § 7–2–3–2–1 to include a predicate definition consisting of more than one clause.

§ 7–2–3-2-1 The Transformation of One Non–Recursive Rule Procedure

The transformation of a predicate definition consisting of one non–recursive rule is done as follows:

Let R be a rule with

a) rule head name r,

b) body predicate names $p_1, p_2, \cdots, p_m$,

c) and variables $V_1, \cdots, V_n$ occurring in the rule head and in at least

one body predicate,

and assume that, each base body predicate (i.e. base table or view) p_i $(1 \leq i \leq m)$ is associated with its data declaration $D_i \in$ normalized-data-declaration, then the transformation of R into a view is as follows:

step 1- The transformation of R body predicate into relational algebraic operations is defined as follows:

a) Since we may have two body predicates with a same predicate name, the name of each body predicate is referred to by an alias. For example, for a given body predicate names $p_1, p_2, \ldots, p_m$, (note p_i and p_j are two body predicates with predicate name), the following expressions are produced:

p_1 is as b_1
p_2 is as b_2
.....
p_m is as b_m

The introduction of the new names is done for sake of clearity and non-ambiguity.

b) For each base body predicate containing one or more constants, each constant is represented by the conjunction of arithmetic expressions. Moreover, these expressions are used as conditions in a SELECT operation. The SELECT operator on relation (i.e. base body predicate) p_m selects tuples from p_m, where each tuple satisfies the arithmetic expressions. For example, given a

relation p_m, and c1 and c2 are constants appearing as g^{th} and k^{th} arguments in p_m, then p_m is transformed into

$$\text{SELECT } p_m \text{ WHERE } p_m.a_g = c1 \wedge p_m.a_k = c2$$

where a_k and a_g are the k^{th} and the g^{th} attribute names, respectively, in p_m, and c1 and c2 are constants.

c) Each two base body predicates, which share variables, are joined together by using =-join operation. For example, if a given variable V appears in k^{th} column of p_i and g^{th} column of p_j, then p_i and p_j are transformed to follows:

$$(p_i \text{ JOIN } p_j) \text{ WHERE } p_i.a_k = p_j.a_g$$

where a_k is the k^{th} attribute name in p_i, and a_g is the g^{th} attribute name in p_j. Each two of the resulting relations or base body predicates (i.e. base body predicates which are not yet joined), which share variables, are joined together by using =-join operation too. For example, if two variables V_1, and V_2 appear in the k^{th} and g^{th} columns of p_i, V_1 appears in the m^{th} column of p_j, V_2 appear in the n^{th} column of p_z, p_i, p_j, and p_z are base body predicates, then they are transformed into RAE as follows:

$$\begin{aligned}&((p_i \text{ JOIN } p_j \text{ WHERE } p_i.a_k = p_j.a_m)\\ &\text{JOIN } p_z \text{ WHERE } p_i.a_g = p_z.a_n)\end{aligned}$$

where a_k and a_g are attribute names in p_i, a_m is an attribute name in p_j, and a_n is an attribute name in p_z. The join of the

resulted relations are carried out until there are no relation sharing variables with an other.

d) The resulted relations form (c) above, which do not share variables, are combined by using the cartesian product operation. For example, given p_i and p_j base body predicates or new relations without shared variables, then they are combined using the cartesian product operation as follows:

$$p_i \text{ TIMES } p_j$$

e) Finally, each comparison predicate is transformed into a similar arithmetic comparison operation which is used as a condition in a SELECT operation on the resulting relation from (d) above. For example, given a comparison body predicate X > Y, where X is j^{th} p_i argument and Y is k^{th} p_g argument, then we can transform the comparison predicate as follows:

$$\text{SELECT p WHERE } p_i.a_j > p_g.a_k$$

where a_j is the j^{th} p_i attribute name and a_k is the k^{th} p_g attribute name, p is a resulting relation from (d) above. Moreover, if one of the operands is a constant, then the comparison predicate is transformed into an arithmetic comparison operation containing a constant as its operand.

step 2- The rule head of R is translated as follows:

$$\text{DEFINE VIEW r } (a_1, \ldots, a_n) \text{ AS PROJECT } p_g.a_1, \ldots, p_k.a_n \text{ WHERE } \Psi$$

where Ψ is an expression built up form step 1, and $(\forall i: 1 \leq i \leq n)$ a_i is V_i corresponding attribute names deduced from the body predicates.

§ 7–2–3–2–2 The Transformation of a Procedure Consisting of More Than One Clause

In this sub–section, we discuss the case when a predicate definition consists of more than one clause. The transformation of a predicate definition consisting of rules and facts is defined as follows: Let P be a predicate definition consisting of clauses (i.e. facts and rules) $C_1, \ldots, C_n$ with the same facts and rules name c.

Each fact C_i $(1 \leq i \leq n)$ in P is transformed into RAE as follows:

1) C_i fact name is replaced by a new fact name f. Moreover, the corresponding fact name in normalized–data–declaration is replaced by f.

2) The fact f data–declaration and its corresponding facts are transformed into RAEs as shown in § 7–2–3–1. The latter is executed.

Each rule C_i $(1 \leq i \leq n)$ is a rule with

a) rule head name c,

b) and body predicate $p_{i1}, \ldots, p_{im}$,

is transformed into RAE as follows:

1) For each rule C_i , the rule head name c is replaced by a new name r_i.

2) The transformation of each rule with rule head name r_i into RAE is defined in § 7–2–3–2–1.

All tables generated from rule transformations and fact transformations, above, are joined together by UNION operation, and then the predicate named c is transformed into RAE as follows:

DEFINE VIEW c AS UNION f, $r_1, \ldots, r_h$

where f is defined in the facts transformation above and $r_1, \ldots, r_h$ are defined in rules transformation above.

For example, rules in Appendix B are translated as shown in Appendix C, by using normalized–data–declaration in figure 7–7.

§ 7–3 The Compilation of TPROLOG Goals

TPROLOG program goals are syntactically similar to C–PROLOG goals (q.v. Appendix A). Therefore, there is no need to translate TPROLOG goals into PROLOG form. However, since TPROLOG programs are extended to include type information (q.v. § 3–3), TPROLOG goals should be typed checked before they are compiled into RAE. Moreover, they should be safety checked too. We use the following example throughout this section:

```
?-glaswegian_infant(L, F, A),
    glaswegian_emp(L,_, school(edinburgh_university, edinburgh), Y).
```

Figure 7–11 A goal in TPROLOG form.

§ 7–3–1 The Transformation of TPROLOG Goals into Complex–Free PROLOG Goals

The transformation of TPROLOG goals into complex–free PROLOG goals consists of three procedures. They are: check_goal_data_type, check_golas_safety, and normalize_goals. The execution of the last procedure depends on the results of the first two procedures.

§ 7–3–1–1 Goals Data Type Checking

Check_goals_data_type (data–declaration–set, data–type–set, goals) = Ok

Check_goals_data_type is a decision procedure. It decides, depending on the data–declaration–set (q.v. § 7–2–1), whether the data of the goal is correctly typed and consistent or not. The procedure takes as an input goals, data–type–set, and data–declaration–set. The result of the decision is defined as follows: suppose that D is a data–declaration–set, T is a data–type–set, and Q is a goal, then

a) Q is well–typed goal (i.e. Ok is true), if the type of all its terms are deducible, and consistent with respect to D and T.

b) Otherwise (i.e. Ok is false), Q is not well–typed and as a result of this decision, it is not possible to get the answer for the goal.

For example the goal in figure 7–11 is well–typed and consistent.

§ 7–3–1–2 Goals Safety Checking

Check_goals_safety (goals, safe–path–set) = Ok

Check_goals_safety is a decision procedure. It takes a set of sub–goals as an input. The procedure checks the safety of each sub–goal with respect to the safety of the unified procedures. The result of the decision is defined as follows: Suppose that, Q is a sub–goal and S is a set of safe paths for Q, then

a) Q is safe goal (i.e. Ok is true), if Q is unified with a strongly safe predicate definition or if Q is mapped to the safe path.

b) Otherwise, Q is unsafe and as a result of this decision there is no need to try to find the answer of the sub–goal.

For example, the goal in figure 7–11 is safe because the unified rules, in figure 7–1, are strongly safe.

§ 7–3–1–3 Normalizing Goals

Normalize_goals (goal, data–type–set, data–type–declaration–set, normalized–data–set)
=(normalized–query, temporary–facts–set)

Normalize_goals takes as an input goals, data–type–set, data–declaration–set, and normalized–declaration–set. It outputs a new–goals–set. A new–goals–set is the normalized form of the goals. The transformation of the goals is defined as follows: Suppose that, Q is a goal, D is a data–declaration–set, and D' is a normalized–declaration–set, then

a) If Q contains a variable of a complex data–type, then Q is normalized in the similar way to its data-declaration normalization (q.v. § 7–2–2–1). However, instead of having a constant of skolem type, we have the variable of a skolem type.

b) If Q contains complex terms, then its normalization is similar to a rule head normalization (q.v. § 7–2–2–3–2). However, the new predicate, which replaces the complex terms, is added to the program as a temporary fact.

For example, the normalized form of the goal in figure 7–11 is shown in figure 7–12. Note, rschool(edinburgh_university, edinburgh) is asserted as a temporary fact.

```
?- glaswegian_infant(L, F, A),
     assert(rschool(edinburgh_university, edinburgh, c)),
     glaswegian_emp(L, F, c, Y).
```

Figure 7–12. The normalization form of goals in figure 7–11.

§ 7–3–2 Transform Complex–Free Goals into RAE

§ 7–3–1 describes how TPROLOG goal are translated into complex–free goals. The transformation procedure results in:

1) A set of temporary facts.
and
2) A set of normalized goals.

This section shows how relational database expressions can be constructed from the above components. The construction is done as follows:

a) All temporary facts are transformed into base–tables as shown in § 7–2–3–1.

b) The normalized goal is transformed into RAE in the similar way to the transformation of rule body predicates. However, a projection operation is used to project the value of all variables in the goal.

For example, the normalized goal in figure 7–12 is transformed into the following expression:

```
PROJECT
      glaswegian_infant.last_name,
      glaswegian_infant.first_name,
      glaswegian_infant.person_age,
      glaswegian_emp.degree_year,
WHERE ( SELECT(glaswegian_infant JOIN glaswegian_emp
            WHERE  glaswegian_infant.last_name = glaswegian_emp.last_name)
            WHERE glaswegian_emp.a2 = c)
```

Since a query is originally written in PROLOG, the result of the query should be in PROLOG form. For example, the result of the query above is rewritten in PROLOG form as follows:

```
L = value(glaswegian_infant.last_name)
F = value(glaswegian_infant.first_name)
A =value(glaswegian_infant.person_age)
Y = value(glaswegian_emp.degree_year)
```

§ 7–4 Status of the Implementation

§ 7–2 and § 7–3 give the plan of the whole system implementation. However, we have not implemented the whole system.

We have implemented the TPROLOG parser which checks the syntax of a TPROLOG program and then translates it into four data set represented in PROLOG form (q.v. § 7–2–1–1–1). After that, the type checker of rules and the typed deduction are implemented (q.v. § 7–2–1–1–2). The fact base normalization has been implemented. This includes the normalization of data–declaration–set (q.v. § 7–2–2–1), the facts type checker, and the normalization of fact–set (q.v. § 7–2–2–2). Finally, we have implemented the rule/goal graph of the program, and then the safety checker is implemented (q.v. § 7–2–2–3–1).

Rules in a TPROLOG program have not been implemented yet. The compilation of the normalized TPROLOG program, which is based on [Reiter 1978, Henschen 1984, Bancilhon 1986, Chang 19986], has not been implemented yet. Finally, the complete processing of TPROLOG goals, which is described in § 7–3, needs to be implemented.

Chapter 8: Conclusions and Future Work

§ 8–1 Conclusions

We may divide our work on compiling logic programs into conventional RAEs into two parts: the pre–compilation and the compilation.

The pre–compilation part is used to check the typing and the safety of a logic program (i.e. a PROLOG program). It ensures the existence of an equivalent RAE for the logic program.

The type system in PROLOG (i.e. TPROLOG) allows us to add type information to PROLOG. It is used to check the correctness and the consistency of the data with respect to the data type information. Moreover, it allows us to get a rich typing system as well as the benefits of PROLOG flexibility in the information representation. However, it does not include recursive type definitions. It allows any term to be of a variant type, whilst database systems do not. We include variant type by assuming that any argument of variant type is a complex argument, and then it is normalize it as shown in the usual way.

Checking the safety of rules by using magic basis [Zaniolo 1986] is restricted to non–recursive PROLOG program rules, whilst the combination of rule/goal graph [Ullman 1985] and magic basis enables us to check the safety of a PROLOG program containing a recursive rules. Although our safety checking is a compile time checking, some safety checking may done at execution time too. We check at compile time

whether the program is strongly safe or weakly safe, whilst at execution time we check whether a query is safe or not. All safety checking is done with respect to a generated rule/goal graph which represents all possible executions of a PROLOG program. The safety conditions discard any unsafe part of the rule/goal graph. Therefore, it would be much faster and economical to incorporate the safety checking into the generation of the graph.

The compilation part compiles logic programs containing non-flat clauses into input suitable for conventional relational database management systems. This is achieved by removing complex arguments from facts and rules and replacing them with simplified facts and rules. The simplified facts are stored in a conventional relational database, and the simplified rules are compiled into views and stored in a rule base. Moreover, temporary facts, which are generated and used at at execution time, are stored temporarily in a database. The removal of complex arguments in this way has several advantages :

1) It enables conventional relational databases to be used for storing the complex facts as ground clauses containing atomic clauses.

2) Standard relational algebraic operations can be used and need not be extended.

3) It allows us to use both logic programming languages and database query languages (e.g. SQL [Chang 1986]).

4) It allows us to use already existing methods of compiling flat clauses into a relational database.

We have implemented a part of the system using C-PROLOG. We have implemented TPROLOG which comprise of translator and the type checker(where translator translates TPROLOG program into PROLOG, and type checker checks the type of the TPROLOG program). Moreover, we have implemented the rule/goal graph generator and the safety checker for a TPROLOG program.

§ 8-2 Future Work

We have the following plan for future work.

1) Extending TPROLOG to include a recursive type definitions. § 3-3-1 allows us to define the type of finite range (i.e. enumerated types, sub-type of the enumerated type or basic type, and structured types which are constructed from finite types or basic types). The inclusion of recursive type definitions would allow us to define in more infinite types (e.g. natural-number).

2) Incorporate the safety conditions into the generation of the graph. A rule/goal graph generator generates a graph of all execution paths of a program, whilst the safety conditions discard unsafe paths. Therefore, we may restricts the generation of the graph to generate only the rule/goal graph containing the unsafe paths. Therefore, if there is no graph generated for the program, then the program is strongly safe. Otherwise it is weakly safe.

3) Complete the whole system and use it with of a real database. This will allow us to experiment and determine more precisely the benefits of combining LPLs and RDBSs.

4) Embedding RAEs in PROLOG. Since not every rule in a PROLOG program can be translated into RAEs. However, some body predicates in a such rule is unified with a procedure which is translated into RAEs. Therefore, we may need to extend PROLOG to include some built–in predicates, such as those introduced by Chang [Chang 1986] which can be used as a bridge to a relational system.

5) Using the RAEs optimization techniques and parallel procedures to execute them. Our approach of translating a PROLOG program rule into RAE is by transforming each base body predicate contains constants into a SELECT operation, and transforming each pair of base predicates into a =join operation and so on until no more relations may be joined. After that, transforming the resulting relations from =join operations to cartesian product operations. Finally, transforming any comparison body predicate into SELECT operation on the top of the resulted relation from the cartesian product operation. We can execute the SELECT operation for each individual base body predicate in parallel. = join operations for each pair of relations may be executed in parallel too. Finally, cartesian product operations for each pair of relations may be executed in parallel. We may impose some optimization techniques which reorder the operations order to make queries more efficient.

References

[Atkinson 1987]

M.P. Atkinson, and O.P. Buneman, " Types and Persistence in Database Programming Languages ", ACM Computing Surveys, Vol. 19, No. 2, June 1987. PP. 105–190.

[Bancilhon 1985]

F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, " Magic Sets and Other Strange Ways to Implement Logic Programs ", MCC Tech. Report No. DB–121–85.

[Bancilhon 1986]

F. Bancilhon, and R. Ramakrishnan, " An Amateur's Introduction to Recursive Query Processing Strategies ", Proceeding of SIGMOD 86 International Conference on Management of Data, Washington D.C. May 1986. PP. 16–52.

[Bell 1978]

R. Bell, and P. Gray, " An Introduction to Relational Algebra: An informal Guide to the ASTRJD Relational Language " University of Aberdeen Tech. Report AUCS/78003, September 1978.

[Bocca 1986]

J. Bocca, " On the Evaluation Strategy of Educe ",Proceeding of SIGMOD 86 International Conference on Management of Data, Washington, D.C. May, 1986, PP. 368–378.

[Bocca 1989]

J. Bocca, " EDUCE* –A Logic Programming System for Implementing KBMS's ", BNCOD–7, Heriot–Watt University, Edinburgh, 12–14 July 1989, PP. 117–146.

[Brodie 1986]

M.L. Brodie, and M. Jarke, " On Integrating Logic Programming and Databases ", Proceeding of 1st International Workshop in Expert Database Systems , 1986, PP. 191–207.

[Bundy 1983]

A. Bundy, " The Computer Modelling of Mathematical Reasoning ", Academic Press Inc, 1983.

[Chang 1986]

C. Chang and A. Walker, " PRSQL : A Prolog Programming Interface with SQL/DS ", Proceeding of 1st International Workshop in Expert Data Base Systems, 1986, PP. 233–246.

[Clocksin 1984]

W.F. Clocksin and C.S. Mellish," Programming in Prolog ", Springer–Verlag Second Edition 1984.

[Date 1981]

C.J. Date, " An Introduction to Database Systems ", Addison–Wesley Publishing Company (Third Edition), 1981.

[Frost 1986]

R.A. Frost, " Introduction to Knowledge Base Systems ", Collins 1986.

[Gallaire 1984]

H. Gallaire, J. Minker and J. Nicolas, " Logic and Data Base: A Deductive Approach ", Computing Survey, Vol. 16, No. 2, June 1984, PP. 153–185.

[Gray 1984]

P. Gray, " Logic, Algebra and Database ", Ellis Horwood Limited, 1984.

[Henschen 1984]

L.J. Henschen and S.A. Naqvi, " On Compiling Queries in Recursive First–Order Databases ", JACM, Vol. 31, No. 1, January 1984, PP. 47–85.

[Hsiang 1984]

J. Hsiang and M.K. Srivas, " A Prolog Environment for Developing and Reasoning About Data Types ", In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, LNCS, Vol 186, TAPSOFT Conference, Berlin 1985, PP. 276–293.

[Jarke 1984]

M. Jarke, J. Clifford, and Y. Vassiliou, " An Optimizing Prolog Front–End To Relational Query System ", 1984, Proceeding of ACM–SIGMOD International Conference on Management of Data, Boston, MA, PP. 296–306.

[Kent 1983]

W. Kent, " A Simple Guide to Five Normal Forms in Relational Data Base Theory " CACM, Vol. 26, No. 2, February 1983, PP. 120–125.

[Kowalski 1975]

R. Kowalski, " A Proof Procedure Using Connecting Graphs ", JACM, Vol. 22, No. 4, 1975, PP. 572–595.

[Kowalski 1979]

R. Kowalski, " Logic Programming for Problem Solving ", North Holland 1979.

[Krishnamurthy 1988]

R. Krishnamurthy, O. Shmueli, and R. Ramakrishnan, " A Framework for Testing Safety and Effective Computability of Extended Datalog " Proce. of SIGMOD International Conference On Management of Data, Chicago June 1988, PP. 154–163.

[Lloyd 1984]

J.W. Lloyd, " Foundation of Logic Programming ", Springer–Verlag, Second Edition 1984.

[Minker 1978]

J. Minker, " An Experimental Relational Database System Based on Logic (or Clause Encounters of a Logical Kind) ", In H. Gallaire and J. Minker, " Logic and Data bases ", Plenum 1978, PP. 107–145.

[Mycroft 1984]

A. Mycroft, and R. A. O'Keefe, " A Polymorphic Type System for Prolog ", AI, Vol. 23, 1984, PP. 295–307.

[Nussbaum 1989]

M. Nussbaum, " Combining Top–Down and Bottom–Up Computation in Knowledge Based Systems " In L. Kerschberg, " Proceeding of the Second International Conference on Expert Database Systems ", 1989, PP. 273–310.

[Parker 1986]

D. Stott Parker, M. Carey, M. Jarke, and A. Walker, " Logic Programming and Data Base ". Proceeding of 1st International Workshop in Expert Data Base Systems, 1986, PP. 35–48.

[Patrice 1987]

B. Patrice, " Turbo Prolog: an Introduction to Artificial intelligence ", John Wiley 1987.

[Ramakrishnan 1987]

R. Ramakrishnan and F. Bancilhon, " Safety of Recursive Horn Clauses with Infinite Relations ", Proceeding of the 6th ACM–SIGACT–SIGMOD–SIGART Symposium on Principles of Data Base Systems, March 1987, PP. 328–339.

[Reiter 1978]

R. Reiter, " Deductive Question–Answering on Relational Data Bases ". In Gallaire H., and Minker J., " Logic and Data Bases " Plenum 1978, PP. 149–177.

[Todd 1976]

S. Todd, " The Peterlee Relational Test Vehicle– a system overview ". IBM System Journal, Vol. 15, No. 4, 1976, PP. 285–308.

[Tsur 1986]

S. Tsur, and C. Zaniolo, " LDL: A Logic-Based Data-Language ", Proceedings of 12th VLDB, Kyoto, Japan, August 1986, PP. 33–41.

[Ullman 1981]

J.D. Ullman, " Principles of Database Systems ", Computer Science Press (Second Edition), 1981.

[Ullman 1985]

J.D. Ullman, " Implementation of Logical Query Languages for Database ", ACM Transactions on Database Systems, Vol. 10, No. 3, September 1985, PP. 289–321.

[Zaniolo 1985]

C. Zaniolo, " The Representation and Deductive Retrieval of Complex Objects ", Proceeding of 11th VLDB, PP. 458–469.

[Zaniolo 1986]

C. Zaniolo, " Safety and Compilation of Non-Recursive Horn Clauses ", Proceeding of 1st International Conference on Expert Database Systems, 1986, PP. 167–178.

Appendix A: EBNF Specification of TPROLOG Syntax

```
program ::= statement { statement }

statement ::= clause
        | '%' fact–declaration '.'
        | '$' data–type '.'
        | '?' goal'.'

data–type ::= type '(' ( '<' con ','integer ',' integer '>'
                    | '{' atom { ',' atom } '}'
                    | predicate < '(' con ':' type { ',' con ':' type } ')'
                    | '[' ( '{' type { ',' type} '}' | type { ',' type } ) ']'
                    )
                  ')'

type ::= con

clause ::= structure ( '.' | ':-' statements '.' )

goal ::= expressions

expressions::= { expression ( ',' | ';' ) } < expression >

expression ::= structure | compute | '!'

structure ::= predicate < '(' term { ',' term } ')' >
```

predicate ::= con

term ::= integer | var | list | structure | "" { char } ""

compute ::= (var | integer)
('is' (var | integer) operator (var | integer)
| logicop (var | integer)
)

operator ::= '+' |'-' | '*' | '/' | 'mod'

logicop ::= '>' < '=' >
| '=' < '<' >
| '<'
| '\='

list ::= '[' < term < { ',' term } < '|' term >>> ']'

atom ::= integer | con

con ::=lo { char }

integer ::= dig { dig }

var ::= cap { char } | '_'

char ::= lo | cap | dig | '_'

dig ::='0' | '1' | ..... | '9'

lo ::= 'a' | 'b' | ..... | 'z'

cap ::= 'A' | 'B' | ..... | 'Z'

fact–declaration ::= predicate '(' con ':' type { ',' con ':' con } ')'

Note:

Nonterminal = { statement, clause, structure, expression, data–type, goal, compute, term, integer, var, list, char, logicop, operator, type, con, expressions, predicate, fact–declaration, atom}

Terminal = {'.', ':-', ',', ';', '%', '?', '!', '(', ')', '"', '+', '-', '*', '/', 'mod', '=', '>', '<', '\=', '[', ']','{', '}', '_', 'e', '0', '1',.....,'9','a', 'b',.....,'z', 'A', 'B',....., 'Z', ':','|', 'is'}

Symbols

< > (zero or one time)

{ } {zero or more time)

( ) (only one time).

Appendix B : The Normalization of Rules in Figure 7–1

```
new_rule(glaswegian_infant, [LN, FN, Age],
    [person, raddress, rnone, <],
    [[FN, LN, V1, Age], [V2, glasgow, V1], [V2], [Age, 4]]).

new_rule(glaswegian_infant, [LN, FN, Age],
    [person, raddress, rflat, <],
    [[FN, LN, V1, Age], [V2, glasgow, V1], [_, _, _, V2], [Age, 4]]).

new_rule(glaswegian_infant, [LN, FN, Age],
    [person, raddress, rhouse, <],
    [[FN, LN, V1, Age], [V2, glasgow, V1], [_, _, V2]], [Age, 4]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree1, rlist1, rdegree1, rlist1, rdegree2, rschool, person, raddress, rnone, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V2], [V3, V1], [_, _, V3], [V4, V1 ], [_, _, Sch, Yr, V4],
    [_, _, Sch], [Fn, Ln, V5, _], [V6, glasgow, V5], [V6], [Yr, 1960], [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree1, rlist1, rdegree1, rlist1, rdegree2, rschool, person, raddress, rflat, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V2], [V3, V1], [_, _, V3], [V4, V1 ], [_, _, Sch, Yr, V4],
    [_, _, Sch], [Fn, Ln, V5, _], [V6, glasgow, V5], [_, _, _, V6], [Yr, 1960], [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree1, rlist1, rdegree1, rlist1, rdegree2, rschool, person, raddress, rhouse, >,
    <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V2], [V3, V1], [_, _, V3], [V4, V1 ], [_, _, Sch, Yr, V4],
    [_, _, Sch], [Fn, Ln, V5, _], [V6, glasgow, V5], [_, _, V6], [Yr, 1960], [Yr, 1990]]).
```

```
new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree1, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, person, raddress,
    rnone, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V2], [V3, V1], [_, _, V4, _V3], [_, _, V4], [V5, V1],
    [_, _, Sch, Yr, V5], [_, _, Sch], [Fn, Ln, V6, _], [V7, glasgow, V6], [V7], [Yr, 1960], [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree1, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, person, raddress,
    rflat, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V2], [V3, V1], [_, _, V4, _V3], [_, _, V4], [V5, V1],
    [_, _, Sch, Yr, V5], [_, _, Sch], [Fn, Ln, V6, _], [V7, glasgow, V6], [_, _, _, V7], [Yr, 1960],
    [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree1, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, person, raddress,
    rhouse, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V2], [V3, V1], [_, _, V4, _V3], [_, _, V4], [V5, V1],
    [_, _, Sch, Yr, V5], [_, _, Sch], [Fn, Ln, V6, _], [V7, glasgow, V6], [_, _, V7], [Yr, 1960],
    [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree2, rschool, rlist1, rdegree1, rlist1, rdegree2, rschool, person, raddress,
    rnone, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, _V3, _V2], [_, _, V3], [V4, V1], [_, _, V4], [V5, V1],
    [_, _, Sch, Yr, V5], [_, _, Sch], [Fn, Ln, V6, _], [V7, glasgow, V6], [V7], [Yr, 1960], [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree2, rschool, rlist1, rdegree1, rlist1, rdegree2, rschool, person, raddress,
    rflat, >, <], [[Fn, Ln,_, V1], [V2, V1], [_, _, _V3, _V2], [_, _, V3], [V4, V1], [_, _, V4],
    [V5, V1], [_, _, Sch, Yr, V5], [_, _, Sch], [Fn, Ln, V6, _], [V7, glasgow, V6], [_, _, _, V7],
    [Yr, 1960], [Yr, 1990]]).
```

```
new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree2, rschool, rlist1, rdegree1, rlist1, rdegree2, rschool, person, raddress,
    rhouse, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, _,V3, _,V2], [_, _, V3], [V4, V1], [_, _, V4], [V5, V1],
    [_, _, Sch, Yr, V5], [_, _, Sch], [Fn, Ln, V6, _], [V7, glasgow, V6], [_, _, V7], [Yr, 1960],
    [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, person,
    raddress, rnone, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V3, _, V2], [_, _, V3], [V4, V1], [_, _, V5, _,V4], [_, _, V5],
    [V6, V1], [_, _, Sch, Yr, V6], [_, _, Sch], [Fn, Ln, V7, _], [V8, glasgow, V7], [V8], [Yr, 1960],
    [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, person,
    raddress, rflat, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V3, _, V2], [_, _, V3], [V4, V1],, [_, _, V5, _,V4], [_, _, V5],
    [V6, V1 ], [_, _, Sch, Yr, V6], [_, _, Sch], [Fn, Ln, V7, _], [V8, glasgow, V7], [_, _, _, V8],
    [Yr, 1960], [Yr, 1990]]).

new_rule(glaswegian_emp, [Ln, Fn, Sch, Yr],
    [emp, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, rlist1, rdegree2, rschool, person,
    raddress, rhouse, >, <],
    [[Fn, Ln,_, V1], [V2, V1], [_, _, V3, _, V2], [_, _, V3], [V4, V1], [_, _, V5, _,V4], [_, _, V5],
    [V6, V1], [_, _, Sch, Yr, V6], [_, _, Sch], [Fn, Ln, V7, _], [V8, glasgow, V7], [_, _, V8],
    [Yr, 1960], [Yr, 1990]]).
```

Appendix C: The Transformation of Rules in Appendix B

```
person      is as  b1
raddress    is as  b2
rmone       is as  b3

DEFINE VIEW r1 (last_name, first_name, person_age)
        AS PROJECT b1.last_name, b1.first_name, b1.person_age
           WHERE SELECT((b1 JOIN b2 WHERE b1.a3= b2.a3)
                        JOIN b3 WHERE b1.a1 = b3.a1)
                 WHERE
                    b2.city_address = glasgow ∧
                    b1.person_age> 4

person      is as  b1
raddress    is as  b2
rflat       is as  b3

DEFINE VIEW r2 (last_name, first_name, person_age)
        AS PROJECT b1.last_name, b1.first_name,b1.person_age
           WHERE SELECT((b1 JOIN b2 WHERE b1.a3= b2.a3)
                        JOIN b3 WHERE b1.a1 = b3.a1)
                 WHERE
                    b2.city_address = glasgow ∧
                    b1.person_age> 4
```

```
person      is as  b1
raddress    is as  b2
rhouse      is as  b3

DEFINE VIEW r3 (last_name, first_name, person_age)
        AS PROJECT b1.last_name, b1.first_name,b1.person_age
           WHERE SELECT((b1 JOIN b2 WHERE b1.a3= b2.a3)
                              JOIN b3 WHERE b1.a1 = b3.a1)
                    WHERE
                       b2.city_address = glasgow ∧
                       b1.person_age> 4

DEFINE VIEW glaswegian_infant AS UNION r1, r2, r3
```

```
emp        is as  b1
rlist1     is as  b2
rdegree1   is as  b3
rlist1     is as  b4
rdegree1   is as  b5
rlist1     is as  b6
rdegree2   is as  b7
rschool    is as  b8
person     is as  b9
raddress   is as  b10
rmone      is as  b11

DEFINE VIEW r4 (last_name, first_name, a2, degree_year)
      AS PROJECT b1.last_name, b1.first_name, b7.a2, b7.degree_year
           WHERE SELECT((((( (((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                  JOIN b3 WHERE b2.a4 = b3.a4)
                                   JOIN b4 WHERE b1.a1 = b4.a1)
                                   JOIN b5 WHERE b4.a4 = b5.a4)
                                   JOIN  b6 WHERE b1.a1 = b6.a1)
                                   JOIN b7 WHERE b6.a4 = b7.a4)
                                   JOIN b8  WHERE b6.a2 =b8.a2)
                             JOIN  b9 WHERE b1.first_name = p9.first_name  ∧
                                                 b1.last_name =  b9.last_name)
                                JOIN (SELECT b10 WHERE
                                  b10.city_address = glasgow)
                                    WHERE  b9.a3 = b10.a3)
                                JOIN b11 WHERE b10.a1 = b11.a1)
            WHERE b7.degree_year > 1960 ∧
                    b7.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree1   is as  b3
rlist1     is as  b4
rdegree1   is as  b5
rlist1     is as  b6
rdegree2   is as  b7
rschool    is as  b8
person     is as  b9
raddress   is as  b10
rflat      is as  b11

DEFINE VIEW r5 (last_name, first_name, a2, degree_year)
      AS PROJECT b1.last_name, b1.first_name, b7.a2, b7.degree_year
           WHERE SELECT((((( (((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                  JOIN b3 WHERE b2.a4 = b3.a4)
                                   JOIN b4 WHERE b1.a1 = b4.a1)
                                   JOIN b5 WHERE b4.a4 = b5.a4)
                                   JOIN  b6 WHERE b1.a1 = b6.a1)
                                   JOIN b7 WHERE b6.a4 = b7.a4)
                                   JOIN b8  WHERE b6.a2 =b8.a2)
                             JOIN  b9 WHERE b1.first_name = p9.first_name  ∧
                                                 b1.last_name =  b9.last_name)
                               JOIN (SELECT b10 WHERE
                                  b10.city_address = glasgow)
                                    WHERE  b9.a3 = b10.a3)
                              JOIN b11 WHERE b10.a1 = b11.a1)
            WHERE b7.degree_year > 1960 ∧
                    b7.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree1   is as  b3
rlist1     is as  b4
rdegree1   is as  b5
rlist1     is as  b6
rdegree2   is as  b7
rschool    is as  b8
person     is as  b9
raddress   is as  b10
rhouse     is as  b11

DEFINE VIEW r6 (last_name, first_name, a2, degree_year)
      AS PROJECT b1.last_name, b1.first_name, b7.a2, b7.degree_year
          WHERE SELECT(((((  (((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                  JOIN b3 WHERE b2.a4 = b3.a4)
                                   JOIN b4 WHERE b1.a1 = b4.a1)
                                   JOIN b5 WHERE b4.a4 = b5.a4)
                                   JOIN  b6 WHERE b1.a1 = b6.a1)
                                   JOIN b7 WHERE b6.a4 = b7.a4)
                                   JOIN b8 WHERE b6.a2 =b8.a2)
                             JOIN  b9 WHERE b1.first_name = p9.first_name  ∧
                                                 b1.last_name =  b9.last_name)
                               JOIN (SELECT b10 WHERE
                                  b10.city_address = glasgow)
                                    WHERE b9.a3 = b10.a3)
                               JOIN b11 WHERE b10.a1 = b11.a1)
           WHERE b7.degree_year > 1960 ∧
                   b7.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree1   is as  b3
rlist1     is as  b4
rdegree2   is as  b5
rschool    is as  b6
rlist1     is as  b7
rdegree2   is as  b8
rschool    is as  b9
person     is as  b10
raddress   is as  b11
rmone      is as  b12

DEFINE VIEW r7 (last_name, first_name, a2, degree_year)
      AS PROJECT b1.last_name,b1.first_name,  b8.a2, b8.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                              JOIN b3 WHERE b2.a4 = b3.a4)
                               JOIN b4 WHERE b1.a1 = b4.a1)
                               JOIN b5 WHERE b5.a4 = b5.a4)
                               JOIN b6 WHERE b52.a2 =b6.a2)
                               JOIN  b7 WHERE b1.a1 = b7.a1)
                               JOIN b8 WHERE b7.a4 = b8.a4 )
                               JOIN b9 WHERE b8.a2 =b9.a2)
                        JOIN  b10 WHERE b1.first_name = b10.first_name ∧
                                        b1.last_name =  b10.last_name)
                           JOIN (SELECT b11 WHERE
                           b11.city_address = glasgow)
                               WHERE b10.a3 =b11.a3)
                           JOIN b12e WHERE b11s.a1 = b12.a1)
               WHERE b8.degree_year > 1960∧
                     b8.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree1   is as  b3
rlist1     is as  b4
rdegree2   is as  b5
rschool    is as  b6
rlist1     is as  b7
rdegree2   is as  b8
rschool    is as  b9
person     is as  b10
raddress   is as  b11
rflat      is as  b12

DEFINE VIEW r8 (last_name, first_name, a2, degree_year)
      AS PROJECT b1.last_name, b1.first_name,  b8.a2, b8.degree_year
            WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                    JOIN b3 WHERE b2.a4 = b3.a4)
                                     JOIN b4 WHERE b1.a1 = b4.a1)
                                     JOIN b5 WHERE b5.a4 = b5.a4)
                                     JOIN b6 WHERE b52.a2 =b6.a2)
                                     JOIN  b7 WHERE b1.a1 = b7.a1)
                                     JOIN b8 WHERE b7.a4 = b8.a4 )
                                     JOIN b9 WHERE b8.a2 =b9.a2)
                               JOIN  b10 WHERE b1.first_name = b10.first_name ∧
                                                   b1.last_name =  b10.last_name)
                                  JOIN (SELECT b11 WHERE
                                  b11.city_address = glasgow)
                                      WHERE b10.a3 =b11.a3)
                                  JOIN b12e WHERE b11s.a1 = b12.a1)
                WHERE b8.degree_year > 1960∧
                        b8.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree1   is as  b3
rlist1     is as  b4
rdegree2   is as  b5
rschool    is as  b6
rlist1     is as  b7
rdegree2   is as  b8
rschool    is as  b9
person     is as  b10
raddress   is as  b11
rhouse     is as  b12

DEFINE VIEW r9 (last_name, first_name, a2, degree_year)
      AS PROJECT b1.last_name, b1.first_name,  b8.a2, b8.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                              JOIN b3 WHERE b2.a4 = b3.a4)
                               JOIN b4 WHERE b1.a1 = b4.a1)
                               JOIN b5 WHERE b5.a4 = b5.a4)
                               JOIN b6 WHERE b52.a2 =b6.a2)
                               JOIN  b7 WHERE b1.a1 = b7.a1)
                               JOIN b8 WHERE b7.a4 = b8.a4 )
                               JOIN b9 WHERE b8.a2 =b9.a2)
                        JOIN  b10 WHERE b1.first_name = b10.first_name ∧
                                        b1.last_name =  b10.last_name)
                           JOIN (SELECT b11 WHERE
                           b11.city_address = glasgow)
                              WHERE b10.a3 =b11.a3)
                           JOIN b12e WHERE b11s.a1 = b12.a1)
               WHERE b8.degree_year > 1960∧
                     b8.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree2   is as  b3
rschool    is as  b4
rlist1     is as  b5
rdegree1   is as  b6
rlist1     is as  b7
rdegree2   is as  b8
rschool    is as  b9
person     is as  b10
raddress   is as  b11
rmone      is as  b12

DEFINE VIEW r10 (last_name, first_name, a2, degree_year)
      AS PROJECT  b1.last_name, b1.first_name,  b8.a2, b8.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                JOIN b3 WHERE b2.a4 = b3.a4)
                                JOIN b41 WHERE b3.a2 =b4.a2)
                                 JOIN b5 WHERE b1.a1 = b5.a1)
                                 JOIN b6 WHERE b5.a4 = b6.a4)
                                 JOIN  b7 WHERE b1.a1 = b7.a1)
                                 JOIN b8 WHERE b7.a4 = b8.a4)
                                 JOIN b9 WHERE b8.a2 = b9.a2)
                           JOIN  b10 WHERE b1.first_name = b10.first_name ∧
                                              b1.last_name =  b10.last_name)
                              JOIN (SELECT b11 WHERE
                              b11.city_address = glasgow)
                                  WHERE b10.a3 = b11.a3)
                              JOIN b12 WHERE b11.a1 = b12.a1)
               WHERE b8.degree_year > 1960∧
                     b8.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree2   is as  b3
rschool    is as  b4
rlist1     is as  b5
rdegree1   is as  b6
rlist1     is as  b7
rdegree2   is as  b8
rschool    is as  b9
person     is as  b10
raddress   is as  b11
rflat      is as  b12

DEFINE VIEW r11 (last_name, first_name, a2, degree_year)
      AS PROJECT  b1.last_name, b1.first_name,  b8.a2, b8.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                JOIN b3 WHERE b2.a4 = b3.a4)
                                JOIN b4l WHERE b3.a2 =b4.a2)
                                 JOIN b5 WHERE b1.a1 = b5.a1)
                                 JOIN b6 WHERE b5.a4 = b6.a4)
                                 JOIN  b7 WHERE b1.a1 = b7.a1)
                                 JOIN b8 WHERE b7.a4 = b8.a4)
                                 JOIN b9 WHERE b8.a2 = b9.a2)
                          JOIN  b10 WHERE b1.first_name = b10.first_name ∧
                                              b1.last_name =  b10.last_name)
                             JOIN (SELECT b11 WHERE
                             b11.city_address = glasgow)
                                 WHERE  b10.a3 = b11.a3)
                             JOIN b12 WHERE b11.a1 = b12.a1)
           WHERE b8.degree_year > 1960∧
                   b8.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree2   is as  b3
rschool    is as  b4
rlist1     is as  b5
rdegree1   is as  b6
rlist1     is as  b7
rdegree2   is as  b8
rschool    is as  b9
person     is as  b10
raddress   is as  b11
rhouse     is as  b12

DEFINE VIEW r12 (last_name, first_name, a2, degree_year)
      AS PROJECT  b1.last_name, b1.first_name,  b8.a2, b8.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                JOIN b3 WHERE b2.a4 = b3.a4)
                                JOIN b4l WHERE b3.a2 =b4.a2)
                                 JOIN b5 WHERE b1.a1 = b5.a1)
                                 JOIN b6 WHERE b5.a4 = b6.a4)
                                 JOIN  b7 WHERE b1.a1 = b7.a1)
                                 JOIN b8 WHERE b7.a4 = b8.a4)
                                 JOIN b9 WHERE b8.a2 = b9.a2)
                          JOIN  b10 WHERE b1.first_name = b10.first_name ∧
                                          b1.last_name =  b10.last_name)
                              JOIN (SELECT b11 WHERE
                              b11.city_address = glasgow)
                                  WHERE b10.a3 = b11.a3)
                              JOIN b12 WHERE b11.a1 = b12.a1)
               WHERE b8.degree_year > 1960∧
                     b8.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree2   is as  b3
rschool    is as  b4
rlist1     is as  b5
rdegree2   is as  b6
rschool    is as  b7
rlist1     is as  b8
rdegree2   is as  b9
rschool    is as  b10
person     is as  b11
raddress   is as  b12
rmone      is as  b13

DEFINE VIEW r13 (last_name, first_name, a2, degree_year)
      AS PROJECT  b1.last_name, b1.first_name,  b9.a2, b9.degree_year
          WHERE SELECT((((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                JOIN b3 WHERE b2.a4 = b3.a4)
                                JOIN b4l WHERE b3.a2 =b4.a2)
                                 JOIN b5 WHERE b1.a1 = b5.a1)
                                 JOIN b6 WHERE b5.a4 = b6.a4)
                                 JOIN b7 WHERE b6.a2 = b7.a2)
                                 JOIN  b8 WHERE b1.a1 = b8.a1)
                                 JOIN b10 WHERE b9.a4 = b10.a4)
                                 JOIN b10 WHERE b9.a2 = b10.a2)
                          JOIN  b11 WHERE b1.first_name = b11.first_name ∧
                                              b1.last_name =  b11.last_name)
                             JOIN (SELECT b12 WHERE
                             b12.city_address = glasgow)
                                 WHERE b11.a3 = b12.a3)
                             JOIN b13 WHERE b12.a1 = b13.a1)
               WHERE b9.degree_year > 1960∧
                       b9.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree2   is as  b3
rschool    is as  b4
rlist1     is as  b5
rdegree2   is as  b6
rschool    is as  b7
rlist1     is as  b8
rdegree2   is as  b9
rschool    is as  b10
person     is as  b11
raddress   is as  b12
rflat      is as  b13

DEFINE VIEW r14 (last_name, first_name, a2, degree_year)
      AS PROJECT  b1.last_name, b1.first_name,  b9.a2, b9.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                JOIN b3 WHERE b2.a4 = b3.a4)
                                JOIN b4l WHERE b3.a2 =b4.a2)
                                 JOIN b5 WHERE b1.a1 = b5.a1)
                                 JOIN b6 WHERE b5.a4 = b6.a4)
                                 JOIN b7 WHERE b6.a2 = b7.a2)
                                 JOIN  b8 WHERE b1.a1 = b8.a1)
                                 JOIN b10 WHERE b9.a4 = b10.a4)
                                 JOIN b10 WHERE b9.a2 = b10.a2)
                       JOIN  b11 WHERE b1.first_name = b11.first_name ∧
                                          b1.last_name =  b11.last_name)
                           JOIN (SELECT b12 WHERE
                           b12.city_address = glasgow)
                               WHERE b11.a3 = b12.a3)
                           JOIN b13 WHERE b12.a1 = b13.a1)
            WHERE b9.degree_year > 1960∧
                   b9.degree_year < 1990
```

```
emp        is as  b1
rlist1     is as  b2
rdegree2   is as  b3
rschool    is as  b4
rlist1     is as  b5
rdegree2   is as  b6
rschool    is as  b7
rlist1     is as  b8
rdegree2   is as  b9
rschool    is as  b10
person     is as  b11
raddress   is as  b12
rhouse     is as  b13

DEFINE VIEW r15 (last_name, first_name, a2, degree_year)
      AS PROJECT  b1.last_name, b1.first_name,  b9.a2, b9.degree_year
          WHERE SELECT((((((((((b1 JOIN b2 WHERE b1.a1 = b2.a1)
                                JOIN b3 WHERE b2.a4 = b3.a4)
                                JOIN b4l WHERE b3.a2 =b4.a2)
                                 JOIN b5 WHERE b1.a1 = b5.a1)
                                 JOIN b6 WHERE b5.a4 = b6.a4)
                                 JOIN b7 WHERE b6.a2 = b7.a2)
                                 JOIN  b8 WHERE b1.a1 = b8.a1)
                                 JOIN b10 WHERE b9.a4 = b10.a4)
                                 JOIN b10 WHERE b9.a2 = b10.a2)
                           JOIN  b11 WHERE b1.first_name = b11.first_name ∧
                                          b1.last_name =  b11.last_name)
                              JOIN (SELECT b12 WHERE
                              b12.city_address = glasgow)
                                  WHERE  b11.a3 = b12.a3)
                              JOIN b13 WHERE b12.a1 = b13.a1)
              WHERE b9.degree_year > 1960∧
                     b9.degree_year < 1990

DEFINE VIEW galswegian_emp AS UNION r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15
```