Applicative Expressions for Relational Programming

by

Husain Ibrahim Haji Ibrahim
B.Sc., King Abdul Aziz University, 1990

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

ACCEPTED
JLTY OF GRADUATE STUDIES
DEAN

We accept this thesis as conforming
to the required standard

Dr. M.H. van Emden, Supervisor (Dept. of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Dept. of Computer Science)

Dr. W.W. Wadge, Departmental Member (Dept. of Computer Science)

Dr. C.G. Morgan, External Examiner (Dept. of Philosophy)

© Husain Ibrahim Haji Ibrahim, 1992

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part,
by photocopy or other means, without the permission of the author.

QA76.62
I27

Supervisor: Dr. Maarten H. van Emden

ABSTRACT

Functional programming comes in two flavors: one where "functions are first-class citizens" (we call this *applicative*) and one based on equations (we call this *declarative*). In relational programming, clauses play the role of equations. Hence pure Prolog is declarative. This thesis provides a basis for the relational analog of applicative functional programming.

We provide a new concept of *table*. We define tables and relations with operators sufficient to translate Horn clauses into applicative form, and establish basic mathematical properties of these operators.

We investigate applicative relational programming, and prove the existence of the least model for our applicative transformations of Horn-clause programs. We show how relations can be first-class citizens, using first-order relational variables and expressions. We devise mechanisms for modularity and genericity, and for local scoping of predicates.

Examiners: ____________________

Dr. M.H. van Emden, Supervisor (Dept. of Computer Science)

____

Dr. M.H.M. Cheng, Departmental Member (Dept. of Computer Science)

____

Dr. W.W. Wadge, Departmental Member (Dept. of Computer Science)

____

Dr. C.G. Morgan, External Examiner (Dept. of Philosophy)

Table of Contents

Title Page i

Abstract ii

Table of Contents iii

List of Tables iv

List of Figures v

1 Introduction 1

1.1 Applicative relational programming 2

1.2 An intuitive view of Horn clauses 4

1.3 Overview 7

2 Tables and relations 8

2.1 Relations and tables in logic programming 8

2.1.1 Tables can be viewed as cylinders 9

2.1.2 Definitions of tables and relations: the difference 10

2.1.3 Solved forms 13

2.1.4 Tables in logic programming 17

2.2 Relational intersection 21

2.3 Relational application: transforming relations to tables 27

2.4 Relational projection: from tables to relations 29
2.5 Invertibility of "/" and ":" 33

3 A basis for relational programming **37**
3.1 Applicative and Horn-clause definitions 38
3.2 Advantages of applicative expressions 45
3.2.1 Modular logic programming 45
3.2.2 Modular relational programming: exporting and importing relations 47
3.2.3 Advanced query language 51

4 Related work **53**

5 Conclusions **56**
5.1 Contributions 56
5.2 Future work 59

Bibliography **63**

A Translation scheme **66**

B Realizing Codd's relational operators **71**

List of Tables

1.1 Classifying functional and relational definitions. 2

2.1 A table for a cylinder. 10

2.2 A table of the answer substitutions of the produce-and-cheese example. 13

2.3 An intersection of two tables. 23

2.4 The table "most like" relation r_1. 27

2.5 $T_2 = r_1 : (a, Y, Z)$. 29

List of Figures

1.1 A geometric interpretation of relational composition: p contains the projection of the intersection of the cylinders on q and r. 5

3.1 The meaning of a relational program. 42

Chapter 1

Introduction

Computer programming can be classified, in general, into three paradigms: state-oriented, functional, and logic programming. The latter two derive their significance from their solid mathematical foundations. Although functional programming preceded logic programming in its development, the latter became an active area of research. In principle, one can argue that this is due to the fact that a function is a special case of a relation [11].

In other words, we see a paradigm of relational programming that is complementary to functional programming, with relations instead of functions as basic entities. We consider logic programming to be the major form of relational programming, along with database theory which has settled into relational form [21, 3].

Still, many problems stand in the way of realizing the ideal of logic programming. The major practical success of logic programming, the Prolog programming language, has theoretical weaknesses. Pure Prolog compromises the completeness of logic programs in order to enhance its feasibility as a programming language. Moreover, its implementations, in standard Prolog systems, introduce many ad-hoc features with a poor, if any, relation

to logic; these clutter the declarative reading of Prolog programs. We argue that the structural deficiencies of logic programming, such as the lack of modularity and block structure, and the lack of higher-order constructs, contributed to these shortcomings.

1.1 Applicative relational programming

Comparing the functional and logic programming paradigms reveals a major gap in the latter; see table 1.1.

Characteristics	functional paradigm	relational paradigm
implicit *element at a time* *declarative*	equations	Horn clauses
explicit *whole function/relation* *applicative*	λ-calculus	?

Table 1.1: Classifying functional and relational definitions.

In functional programming, there are two styles of definitions, one using expressions of the λ-calculus and the other using equations. Take for example the (higher-order) function "*twice*." It can be denoted by the λ-expression "$\lambda f.\lambda x.f(fx)$"; we call this an *applicative* definition. This function can also be defined in a *declarative* style, that is, by asserting as true certain equations, as follows:

```
twice : F = g(F)
g(F) : X = F : (F : X)
```

The characteristics of each style can be summarized in the first two columns of table 1.1. In the applicative style, functions can be arguments

and results of functions, and can be bound to variables. Hence the slogan "functions are first-class citizens" in the applicative style, meaning that they are a standard type of value. This is not the case in the declarative style. As a result, functional expressions are only available in the applicative style.

Furthermore, it can be shown that the two styles have complementary strengths, namely local scoping and modularity in the applicative one, and a natural expression of recursion and selection in the declarative one. Thus it is valuable for a programmer to have both available and to be able to switch effortlessly between the two. In functional programming, the theory of such a combination has been developed and practical applications have been reported [5, 7, 19].

Relational programming has its best developed form, logic programming, exclusively based on Horn clauses, which constitute a declarative paradigm. As a result, the use of expressions is restricted in logic programming. This also explains its lack of program structuring mechanisms. It is essential for any programming language to provide expressions, with values of its standard types, as well as to have program structuring aids. Therefore, the absence of an applicative language in which relations are first-class objects is a major concern for relational programming.

In this thesis, we develop a basis for applicative relational programming that provides for relational expressions so as to obtain alternatives for some of the deficiencies in Prolog. The framework we construct has a first-order logical characterization and a simple intertranslation with Horn clauses. This is achieved by extending the syntax of logic programming to include notations for relations, based on the following intuition of the meaning of Horn clauses.

1.2 An intuitive view of Horn clauses

We first give the intuitive meaning of Horn clauses that motivates our applicative framework. Consider the Horn clause:

$$p(X, Z) \leftarrow q(X, Y), r(Y, Z) \tag{1.1}$$

It can be read declaratively as saying: p holds for X and Z if q holds for X and Y and r holds for Y and Z. We call p a relational composition of q and r.

There are two points of importance here. The first is *how* does the clause define the relation p in its head in terms of the relations q and r in the condition part? The answer to this question reveals the operations in the Horn clause. The second point is that, as suggested by the declarative reading, the head relation p is defined *one element at time.* In the following chapter, we construct an applicative framework with operations oriented towards whole relations. This will be as a generalization for the answer to the first question.

Let us consider the first point. Figure 1.1 gives an exemplary interpretation of clause 1.1 from analytical geometry. The variables X, Y, and Z represent the coordinates. Each of the conditions $q(X, Y)$ and $r(Y, Z)$ is a cylinder in the three coordinates on a base of its respective relation. The conjunction "$q(X, Y), r(Y, Z)$" of these conditions is represented by the intersection of their cylinders. Notice that for the relation denoted by q, only a partial cylinder is shown: the one containing the portion participating in the intersection. This intersection is projected on the subspace spanned by the variables X and Z of the conclusion, giving the part of p defined by the clause.

Thus a definite Horn clause in general can be interpreted to say that: *the relation in the conclusion contains the projection, onto the subspace spanned*

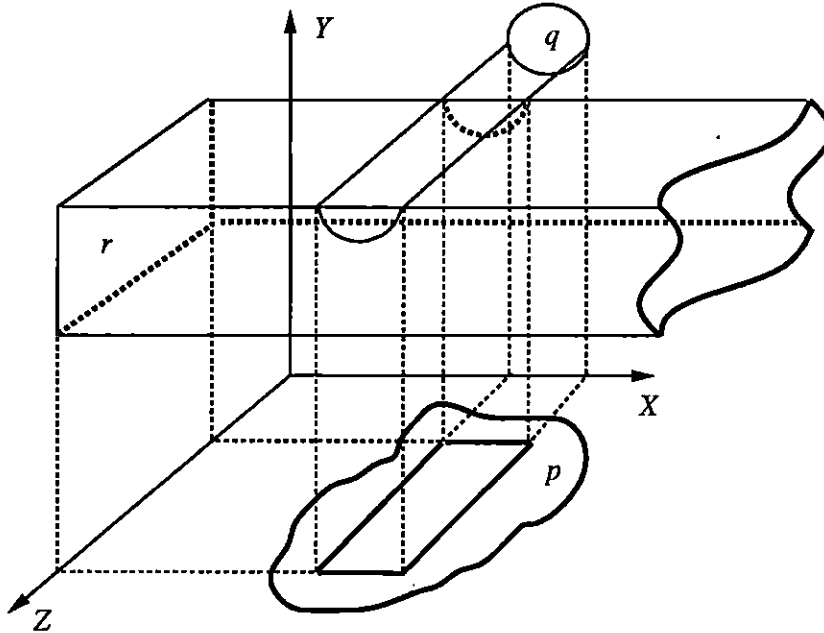


Figure 1.1: A geometric interpretation of relational composition: p contains the projection of the intersection of the cylinders on q and r.

by the variables in the conclusion, of the intersection of cylinders on the relations in the condition part.

Accordingly, a Horn clause may define the relation in its conclusion, as a whole, by means of three operations on relations: cylindrification of the atoms in the condition part, intersection of the resulting cylinders, and projection of that intersection on the set of variables in the conclusion. This suggests that we can transform Horn clauses into an applicative form with operations on relations as a whole. Here, we motivate such a form.

Inspired by the so-called "window principle" of [8] as operational semantics for Prolog, we can derive a tuple in the relation p through three steps. In the first, each of the goals $q(X,Y)$ and $r(Y,Z)$ is satisfied by matching its tuple, through successful unification, with one in its corresponding relation.

The tuples of these relations are derived in the same way, one at a time. We take as answer to a goal the set of bindings, in the form of equations, of the variables in the goal to the corresponding components in the matching tuple. Now, each atom consists of a predicate symbol representing the relation and a tuple of terms as one value in it (or a template for one value, if it contains variables). Hence, intuitively, we can think of an *application* of the relation represented by the predicate, say q, to the tuple, (X, Y), yielding a match, say (a, b), hence an answer, $\{X = a, Y = b\}$.

In the second step, the conjunction "$q(X, Y), r(Y, Z)$" of the conditions is implemented by checking that their answers are consistent with respect to instantiation of the variables in the conjunction. We combine the two answers and apply unification to find a solution, if any, to the resulting system of equations. This solution, as will be seen in chapter 2, is the same as the *composition* of the two answers. For example, if the answers are $\{X = a, Y = b\}$ and $\{Y = B, Z = c\}$, where B is a variable, then we get $\{X = a, Y = b, Z = c\}$ as their composition.

These substitutions are used in the third step to instantiate any corresponding variables in the conclusion. This *abstracts* from the composition and the conclusion a tuple *in* the defined relation p; e.g., the tuple (a, c) in the last example.

These three operations, application, composition, and abstraction, constitute a tuple-at-a-time operational semantics for Horn clauses. To make them oriented towards relations, we generalize these operations in chapter 2.

This way, we motivate a relation-oriented transformation for Horn clauses that corresponds to our intuition of their meaning, with application for cylindrification, composition for intersection, and abstraction for projection.

1.3 Overview

Chapter 2 deals with the development of our applicative framework for Horn clauses. There, we provide *tables* as a notation for the cylinders in our intuition, and integrate them into logic programming in two ways, as goals and as conditions. We define operators on tables and relations and establish their basic mathematical properties.

The feasibility of our framework as a basis for applicative relational programming is demonstrated in chapter 3. We show that:

1. Our operators are sufficient to translate Horn clauses into applicative form, while retaining their least fixpoint characterization.

2. Relations can be first-class objects.

3. Our framework provides local scoping for predicate symbols.

4. Our framework facilitates integrating λ-operators into applicative relational programming. This yields an export/import mechanism for relations between programs that provides for generic and composite modules.

5. The framework is more flexible as a query language than Horn clauses.

Chapter 4 discusses important related work, especially the applicability of our framework to databases, and set abstractions, an alternative method for relational definitions suggested by McCabe [17]. Chapter 5 concludes with a summary of the contributions of this thesis, and an exposition of points to be pursued.

Chapter 2

Tables and relations

In this chapter, we design a framework that allows us to translate the definite Horn clauses of pure Prolog to applicative form.

From the previous chapter we recognize relations and cylinders, a special type of relation, as two types of values. We start by defining these types and introducing the second one into logic programming by means of a new operator ("`where`"). We saw also that Horn clauses suggested three operations on relations, namely application (":"), intersection ("$\cap$"), and projection ("/"). We define each operation in a separate section, along with some basic results. The last section discusses the invertibility of the application and projection operations.

2.1 Relations and tables in logic programming

In logic programming, relations, queries, and programs are the standard types of values. We need to add cylinders as another type of value to provide the right concept for our relational operations.

2.1.1 Tables can be viewed as cylinders

A cylinder, in our intuitive view of the meaning of Horn clauses, is specified by a *base* relation and a selector of variables. The cylinder of the condition $q(X,Y)$, for example, has the relation denoted by q as its base and $\{X,Y\}$ as a selector. Assuming that $q = \{(a,b),(b,c),(c,a)\}$, this cylinder can be specified by the set of substitutions

$$\{\{X = a, Y = b\}, \{X = b, Y = c\}, \{X = c, Y = a\}\},$$

as this specifies that the base relation of the cylinder is $\{(a,b),(b,c),(c,a)\}$ and that the selector is $\{X,Y\}$.

A substitution is a set of equations with distinct variables as their left-hand sides each of which is distinct from the term on the right-hand side. By using equations in substitutions, instead of ordered pairs, we deviate from standard logic programming syntax. This is convenient for the development of our work. Nevertheless, we retain the standard use of substitutions: to apply a substitution θ to a term t, we *simultaneously* substitute all occurrences in t of every variable on the left-hand side of an equation in θ with the corresponding right-hand side; e.g. applying $\{X = Y, Y = b\}$ to X yields Y, not b.

The set of substitutions that we used to denote the above cylinder is of a special type; the substituted variables in each substitution were the same. Such sets of substitutions play a central role in this work. Observe, in the first place, that a table is a natural notation for such a set of substitutions. The right-hand sides make up the rows of the table, while the left-hand sides need not be repeated, and hence can be the headings of the table's columns; see table 2.1. Because of this, we call such sets of substitutions "tables."

X	Y
a	b
b	c
c	a

Table 2.1: A table for a cylinder.

In the second place, observe that the set of answers to a Prolog query is a table. For example, suppose that the logic program P is the set of facts $\{q(a,b), q(b,c), q(c,a)\}$. The query ?- $q(X,Y)$ produces table 2.1 as set of answer substitutions.

Although we have now established in principle the nature of a table, we still need to give our formal definition of a table, and explain the reason for our choice.

2.1.2 Definitions of tables and relations: the difference

Given a set of constants and function symbols and a set of variables, a *term* is a constant or a variable or an expression of the form $f(t_1, \ldots, t_n)$, where f is a given n-ary function symbol and each t_i is a term for $i \in n$. The Herbrand universe on the given sets is the set of all possible variable-free terms.

From a set theoretical point of view, an n-ary relation is a set of n-tuples. We take the *Herbrand universe* as base set for relations.

Definition 1 *An n-ary relation r over a Herbrand universe U_H is a set of n-tuples of elements of U_H; i.e., $r \subseteq (n \rightarrow U_H)$. The natural number "$n$" is the* arity *of the relation.*

Note that the elements of the Herbrand universe are variable-free (or *ground*), so that relations over the Herbrand universe are always ground.

When we add predicate symbols to the given sets of symbols we can consider the set of all atomic formulas $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol and each t_i is a ground term for $i \in n$. This set is the *Herbrand base* on the augmented symbol sets. An *Herbrand interpretation* is a subset of the Herbrand base. Herbrand interpretations are ordered by set inclusion.

For any set of clauses (or a sentence) that can be written using given symbol sets, the following rules, adopted from [14], determine whether the sentence is true in an Herbrand interpretation I for those symbol sets:

1. A sentence S is true in I iff every clause of S is true in I.
2. A clause is true in I iff every variable-free instance of it is true in I.
3. A variable-free clause is true in I iff at least one of its conditions is not in I or at least one conclusion is in I.

If the sentence is true in an Herbrand interpretation, then the interpretation is said to be a (Herbrand)*model* of the sentence. If the sentence contains positive Horn clauses only, then there is among all models a unique *minimal* one, when set inclusion is the ordering among these models [23].

Definition 2 *Let P be a set of positive Horn clauses. By "ν" we denote the function that maps P to its minimal Herbrand model, written as νP.*

To introduce tables, we first show how a notion of table can be naturally based on the answers to a query in logic programming.

With Prolog, we interrogate a program by means of a query. For example, to find each name of an employee in a produce department with the same age as an employee in a cheese department, we enter the Prolog query

```
?- dept(cheese,Dept1),  employee(Emp1,Dept1,Age),
   dept(produce,Dept2), employee(Emp2,Dept2,Age).
```

Prolog returns zero or more answer substitutions for the set of variables in the query through a process known as *SLD-resolution* [15]. Suppose that in this example we obtain the two answer substitutions:

```
Dept1 = 1234, Emp1 = dave, Age = 51, Dept2 = 2341, Emp2 = bill.
Dept1 = 4321, Emp1 = john, Age = 22, Dept2 = 1432, Emp2 = mike.
```

Each of the two answer substitutions is a special case of a set of *term equations*, where each equation has terms of clausal logic on each side. Such a set of equations is in *solved form* if all left-hand sides are variables and if these left-hand sides are the only occurrences of those variables [16]. Hence a set of equations in solved form is a substitution, where each right-hand side substitutes for the corresponding left-hand side.

For example,

```
X = Y, Y = Z, f(X,Z) = f(a,a)
```

is a set of equations that is not in solved form, but it is equivalent to one that is, namely:

```
X = a, Y = a, Z = a.
```

Conversely, a substitution can be regarded as a set of equations in solved form. This is facilitated by our equational syntax for substitutions. In this way we regard the answer substitutions for a Prolog query as a set of sets of equations in solved form, with the additional constraint that the solved forms have a common set of variables as left-hand sides. This constraint makes it natural to present a set of answer substitutions for a given query as a table.

The reason is that these sets of equations can be economically represented in print as a table where the left-hand sides are the headings of the columns and the right-hand sides of each set constitute a row of the table.

For example, table 2.2 represents the answer substitutions of the produce-and-cheese example above.

Dept1	Emp1	Age	Dept2	Emp2
1234	dave	51	2341	bill
4321	john	22	1432	mike

Table 2.2: A table of the answer substitutions of the produce-and-cheese example.

Definition 3 *An n-ary table is a set of sets of equations in solved form. Each set of equations in the table has the same set of n variables as left-hand sides.*

In spite of the similarities, there are important differences between relations and tables as defined here. The main difference is that *the right-hand sides in the equation sets of a table may contain variables while a relation is always ground.* Also, the order does not matter for the columns of a table while it is significant for the arguments of each tuple in a relation if the arity is an ordered set.

It is useful to have a notation for the set of variables as left-hand sides of the equations in a substitution. We denote this set by "τ_θ," where θ is a substitution. We may also use τ_T for a table T since τ_θ is the same for any $\theta \in T$.

2.1.3 Solved forms

For our framework, the solvability of sets of term equations is important. We, therefore, consider them in more detail, along with some related notions.

Definition 4 *A unifier for a term equation $t = t'$ is a substitution θ such that $t\theta$ and $t'\theta$ are syntactically identical. A unifier for a set s of term equations is a substitution θ that is a unifier for each equation $e \in s$.*

Definition 5 *Let s be a set of term equations. By $\psi(s)$ we assert that s is solvable, where $\psi(s) \Leftrightarrow \exists\theta.\, \theta$ is a unifier for s.*

Finding a unifier for a given set of term equations is the unification problem. A unification algorithm shows how the given set can be transformed into an equivalent one in solved form, if any exists. Two equation sets are equivalent if they have the same set of unifiers. Regarded as a substitution, a set θ of equations in solved form is a unifier of itself. Moreover, it is the *most general unifier*[1] (*mgu*), since any other unifier η for θ must at least substitute for all its left-hand side variables. Hence, a unification algorithm determines whether a given set of term equations is solvable, and if it is, the algorithm finds its mgu.

We restrict the solved form, if any, of a set of term equations to its mgu. This makes the solved form unique up to renaming of variables, while capturing all others, as they can be obtained by instantiation.

Definition 6 *Let s be a set of equations such that $\psi(s)$. The solved form of s, denoted by $\phi(s)$, is: $\phi(s) \stackrel{\text{def}}{=}$ the mgu of s.*

A unification algorithm. We state here a unification algorithm due to Martelli and Montanari [16].

Given a set of term equations, repeatedly perform on the current set of equations any of the rules below. If no rule applies, stop with success.

[1] A unifier is most general if any other unifier, for the same terms, can be derived from it by another substitution.

1. Delete any equation of the form $x = x$.

2. Select any equation of the form $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$. If f and g are different functors or $n \neq m$, stop with failure; otherwise, replace this equation with the equations $\{s_1 = t_1, \ldots, s_n = t_n\}$.

3. Select any equation of the form $t = x$, where x is a variable and t is not, and replace it by $x = t$.

4. Select any equation of the form $x = t$ where x is a variable which occurs somewhere else in the set of equations and where $t \neq x$. If x occurs in t, then halt with failure; otherwise, apply the substitution $\{x = t\}$ to all other equations.

The algorithm, as proved by its authors, terminates for every initial set of equations either with failure, when no unifier of the initial set exists, or with success, with the final set of equations being the mgu of the initial set.

We will occasionally require two kinds of restriction. The first is to restrict the unification algorithm so that it solves *for* a specific set of variables; i.e. if the resulting unifier is θ, then these variables should be in τ_θ. For this purpose, we allow such a set as an optional additional parameter for unification. The unification algorithm is then slightly modified as variables occurring in this parameter will have priority over other variables when applying step 4. We denote the result of this restricted unification by $\phi_V(s)$, where s is the initial set of equations and V is the restricting set of variables.

The second restriction concerns the result of unification. Given a substitution θ, we denote its restriction to a specific set V of variables by $\theta[V]$, and define it as $\{x = t \mid x = t \in \theta, x \in V\}$. Thus, $\phi_V(s)[V]$ imposes both restrictions on the unification of the initial set of equations s. Notice that

the last operation easily extends to tables with $T[V]$ for a table T being $\{\theta[V] \mid \theta \in T\}$.

Since $\phi(s)$, if it exists, is equivalent to the set s of term equations, then for any such sets s and t:

$$\psi(s \cup t) \Leftrightarrow \psi(\phi(s) \cup \phi(t)), \text{ and} \tag{2.1}$$

$$\phi(s \cup t) = \phi(\phi(s) \cup \phi(t)). \tag{2.2}$$

Only when $\phi(s)$ and $\phi(t)$ have no common variable that is the left-hand side of one of their equations, rule 2.2 reduces to

$$\phi(s \cup t) = \phi(s) \cup \phi(t). \tag{2.3}$$

Otherwise, $\phi(s) \cup \phi(t)$ is not a solved form, and may not be reducible to one, since the substitutions concerning any common variable may be inconsistent. For example,

$$\phi(\{X = Y, Y = Z\}) = \{X = Z, Y = Z\}, \text{ and}$$

$$\phi(\{f(X, Z) = f(a, a)\}) = \{X = a, Z = a\}, \text{ while}$$

$$\phi(\{X = Y, Y = Z, f(X, Z) = f(a, a)\}) = \{X = a, Y = a, Z = a\}$$
$$= \phi(\{X = Z, Y = Z, X = a, Z = a\} \neq \{X = Z, Y = Z, X = a, Z = a\}.$$

We adopt from [15] the definition of *composition* of substitutions along with a property that we require later. Let $\theta = \{u_1 = s_1, \ldots, u_m = s_m\}$ and $\eta = \{v_1 = t_1, \ldots, v_n = t_n\}$ be substitutions. Then the composition of θ and η, denoted $\theta\eta$, is the substitution obtained from the set

$$\{u_1 = s_1\eta, \ldots, u_m = s_m\eta, v_1 = t_1, \ldots, v_n = t_n\}$$

by deleting any equation $u_i = s_i\eta$ whose two sides are identical and deleting any equation $v_j = t_j$ for which $v_j \in \{u_1, \ldots, u_m\}$.

Proposition 1 *If θ and η are substitutions and t is a term, then* $(t\theta)\eta = t(\theta\eta)$.

2.1.4 Tables in logic programming

To establish the role of tables, as values, in logic programming, we show that tables can be answers to a query on a logic program as well as conditions in a Horn clause.

Tables as results from queries on logic programs. Tables, as defined above, can be obtained in logic programming as follows:

Definition 7 *Let P be a logic program, Q a query, and T an SLD-tree for P and Q. The expression (Q* `where` *P) has as value the table of the answer substitutions associated with all the success leaves of T.*

The symbol "`where`" is an infix binary operator and Q and P are its operands.

An SLD-tree for a program P and a query Q is a tool for specifying the search space of their SLD-derivations. Each of its nodes is a query, which may be empty, and the root is Q. If a node is empty, it has no children; otherwise, for each input clause whose head is unifiable with the selected goal of the node, the node has the resolvent of the goal and the clause as a child. The resolvent is obtained from the node by replacing its selected goal with all the goals in the body of the input clause and applying to this new query the mgu of the selected goal and the head of the input clause.

Hence, each branch of the SLD-tree is a derivation of P and Q, and one that ends with the empty clause is *successful.* The *answer substitution* of a

successful branch is the composition of the mgu's[2] from the root to the leaf node restricted to the variables in the root query.

Notice that this composition, as proved in [15], is an mgu of the combined set of mgu's; i.e. the answer substitution is in solved form. In fact, such an answer substitution is known as a *computed answer* for $\{Q\} \cup P$ [15]. Hence, the value of the table (Q `where` P) is the set of all computed answers for $\{Q\} \cup P$. Of course, as the table is a set, duplicate answers are removed.

Since Q is a query of logic programming and P is a logic program, there are no relational variables in the `where` expression. However, any predicate symbol in Q is bound by P. The individual variables in Q, if any, are bound as they are in logic programming. Yet, since they appear in the value of the `where` expression, they act as free variables. This is reflected when composing tables, whether by intersection or union.

We restate here the main results about computed answers in terms of the value of the `where` expression. These are adopted from [15] and are needed to prove further results.

Lemma 1 *(Soundness)*
$\theta \in (Q$ `where` $P) \Rightarrow \theta$ *is a correct answer for* $\{Q\} \cup P$.

A *correct answer* for a query on a program is a substitution restricted to the variables in the query that, when applied to the conjunction of the atoms in the query, yields a logical consequence of the program.

Lemma 2 *(Completeness)*
For every correct answer θ *for* $\{Q\} \cup P$, $\exists \eta \in (Q$ `where` $P)$ *such that* θ *is an instance of* η*; i.e.,* $\theta = \eta\gamma$ *for some substitution* γ.

[2]We assume eliminating the variables of the initial query first in the unifications resulting in these mgu's.

Lemma 3 *The value of* $(Q$ `where` $P)$ *does not depend on the SLD-tree* T *in definition* 7.

Special tables. The concept of table as result of a query on a logic program leads to two distinguished tables. When a query Q fails for a logic program P in Prolog, there are no answer substitutions and the table $(Q$ `where` $P)$ is empty. Although there are in principle many empty tables, one for each set of variables in Q, we do not distinguish between them and use a special symbol for all of them.

Definition 8 *The table* $\bot \stackrel{\text{def}}{=} \{\}$.

The query Q may have no variables. A successful SLD-derivation starting with such a query results in the empty answer substitution, that is, the empty set of equations, which we regard as a solved form. The table resulting from such a derivation contains this set as its only element. For this table we also use a special symbol:

Definition 9 *The table* $\top \stackrel{\text{def}}{=} \{\{\}\}$.

Tables as conditions in Horn clauses. Note that the condition part of a Horn clause has the same type as a query. If a query evaluates to a table, then so does a condition part. So we should be able to substitute a table for a condition part in a Horn clause. The meaning of such a modified Horn clause readily suggests itself.

Definition 10 *For any atom* A*, any table* T*, and an interpretation*[3] I*,*

$$A \leftarrow T \textit{ is true in } I \stackrel{\text{def}}{\Leftrightarrow} \forall \theta \in T, A\theta \textit{ is true in } I.$$

[3]Unless stated otherwise, interpretation means Herbrand interpretation and model means Herbrand model, throughout the thesis.

The following two theorems justify our definition. The first shows that the definition preserves the meaning of truth of a Horn clause in an Herbrand interpretation when the table is based on the program of that interpretation. The program of an interpretation is defined as follows.

Definition 11 *For any interpretation I, the program of I, denoted P_I, is the program containing as clauses exactly the ground atomic formulas of I.*

Theorem 1 *For all atoms A, $B_1, \ldots, B_n$, and an interpretation I,*

$$A \leftarrow ((\leftarrow B_1, \ldots, B_n) \texttt{ where } P_I) \textit{ is true in } I$$

$$\Leftrightarrow A \leftarrow B_1, \ldots, B_n \textit{ is true in } I.$$

Proof Let C be $A \leftarrow B_1, \ldots, B_n$, and let T be $((\leftarrow B_1, \ldots, B_n) \texttt{ where } P_I)$.

To prove the "if" part of the theorem, assume that C is true in I and let $\theta \in T$. We have $\{B_1\theta, \ldots, B_n\theta\} \subseteq P_I$, by definitions 7 and 11, as θ is a ground answer substitution for $(\leftarrow B_1, \ldots, B_n)$ and P_I. Hence, $B_1\theta, \ldots, B_n\theta$ are true in I. But this implies, by the assumption and by the meaning of a clause true in an interpretation, that $A\theta$ is true in I. Hence, as θ was an arbitrary element of T, $A \leftarrow T$ is true in I follows from definition 10.

To prove the "only if" part, assume that $A \leftarrow T$ is true in I and observe that $B_1\eta, \ldots, B_n\eta \in I$ for ground η implies that $B_1\eta, \ldots, B_n\eta \in P_I$, by definition 11. Hence, η is an answer substitution for $(\leftarrow B_1, \ldots, B_n)$ and P_I so that, by definition 7, $\eta \in T$. Thus, by the assumption and definition 10, $A\eta$ is true in I. As η was selected arbitrarily, $A \leftarrow B_1, \ldots, B_n$ is true in I.
□

The next theorem shows that definition 10 preserves the meaning of truth of a Horn clause in the least model of a program when the table is based on that program.

Theorem 2 *For every logic program P and all atoms $A, B_1, \ldots, B_n$,*

$$A \leftarrow ((\leftarrow B_1, \ldots, B_n) \texttt{ where } P) \textit{ is true in } \nu P$$

$$\Leftrightarrow A \leftarrow B_1, \ldots, B_n \textit{ is true in } \nu P.$$

Proof Let Q be $\leftarrow B_1, \ldots, B_n$. We have $A \leftarrow Q$ is true in νP
$\Leftrightarrow Q\sigma \in \nu P \rightarrow A\sigma \in \nu P$ for any ground substitution σ, by definition of a clause true in an interpretation
$\Leftrightarrow \theta \in (Q \texttt{ where } P) \rightarrow A\theta\gamma \in \nu P$ for some θ and a ground substitution γ such that $\theta\gamma = \sigma$, by lemma 2 for the "only if" part, as σ is correct, and for the "if" part by lemma 1
$\Leftrightarrow \theta \in (Q \texttt{ where } P) \rightarrow A\theta$ is true in νP, by definition of an atom true in an interpretation, since σ and γ are arbitrary
$\Leftrightarrow A \leftarrow (Q \texttt{ where } P)$ is true in νP, by definition 10.
□

2.2 Relational intersection

In Prolog, the answer substitutions to a query with the goals g_1 and g_2 are obtained by combining pairwise the answer substitutions to the separate queries with goal g_1 and with goal g_2. Colmerauer [8] has formulated this combination by means of solved forms of equations, as follows.

Let θ_1 and θ_2 be answer substitutions to the queries ?- g_1 and ?- g_2 respectively. θ_1 and θ_2 are sets of equations in solved form. When g_1 and g_2 have no variables in common, then $\phi(\theta_1 \cup \theta_2) = \theta_1 \cup \theta_2$. When they do have a common variable, $\theta_1 \cup \theta_2$ is not in solved form and may not be reducible to one. Colmerauer observed, in his "window principle", that $\phi(\theta_1 \cup \theta_2)$, if it exists, is an answer substitution for the query ?- g_1, g_2.

For each of the queries ?- g_1 and ?- g_2, the sets of its answer substitutions constitute a table. We generalize Colmerauer's method to define the intersection of these tables. For each θ_1 and θ_2 in the two tables, we include $\phi(\theta_1 \cup \theta_2)$ in the intersection. In other words, the intersection of tables S and T can be defined as

$$S \cap T \stackrel{\text{def}}{=} \{\phi(s \cup t) \mid s \in S, t \in T, \text{ and } \psi(s \cup t)\}.$$

Obviously, we want the result of intersection to be a table. This requires of all the solved forms in the result to have the same set of variables as left-hand sides. An obvious choice of such a set is the union of the variables as headings of the two intersecting tables. Because of the non-deterministic nature of our unification algorithm, the suggested definition, in general, does not guarantee this choice. Moreover, the result of the above intersection may not be a table.

For example, take the two tables $S = \{\{X = a, Z = Y\}, \{X = a, Z = a\}\}$ and $T = \{\{X = Z\}\}$. $S \cap T = \{\{X = a, Z = a, Y = a\}, \{X = a, Z = a\}\}$ which is not a table. We need to restrict the solved forms in the above definition to the union of the sets of variables as headings of the two intersecting tables.

Next, let $R = \{\{X = Y, Z = Y\}\}$. $R \cap T$, where T is as above, could be R as well as $\{\{X = Z, Y = Z\}\}$. Thus, we need to restrict also the unification

in the above definition so that it solves for the variables as left-hand sides of the equation sets in both tables.

Definition 12 *The intersection (denoted $\cap$) of tables S and T is defined as*

$$S \cap T \stackrel{\text{def}}{=} \{\phi_V(s \cup t)[V] \mid s \in S, t \in T, \text{ and } \psi(s \cup t)\},$$

where $V = \tau_S \cup \tau_T$.

In this way, the result of intersection is a table with a heading consisting of the distinct variables in the headings of both operands.

As an example of the intersection of tables, let program P be the set of ground facts $\{q(a,b), q(b,c), q(c,a)\}$. Then the result of the query $((\leftarrow q(X,Y))$ `where` $P) \cap ((\leftarrow q(Y,Z))$ `where` $P)$ is table 2.3.

X	Y	Z
a	b	c
b	c	a
c	a	b

Table 2.3: An intersection of two tables.

The following theorem gives basic properties of relational intersection[4].

Theorem 3 *Intersection is commutative and associative. Intersection has a unique null element, which is $\bot$ and a unique unit element, which is $\top$.*

Proof

1. Commutativity: $S \cap T$

   $= \{\phi(s \cup t) \mid s \in S, t \in T, \text{ and } \psi(s \cup t)\}$, by definition 12

   $= \{\phi(t \cup s) \mid t \in T, s \in S, \text{ and } \psi(t \cup s)\}$, as $\cup$ is commutative

   $= T \cap S$, by definition 12.

[4] From now on, we keep the notation simple by implying the restrictions on the forms $\phi(s \cup t)$ whenever they are in the context of intersection.

2. Associativity: $S \cap (T \cap R)$

   $= \{\phi(s \cup tr) \mid s \in S, tr \in T \cap R,$ and $\psi(s \cup tr)\}$, by definition 12

   $= \{\phi(s \cup \phi(t \cup r)) \mid s \in S, t \in T, r \in R,$ and $\psi(s \cup \phi(t \cup r))\}$, by definition 12 where tr is $\phi(t \cup r)$

   $= \{\phi(s \cup (t \cup r)) \mid s \in S, t \in T, r \in R,$ and $\psi(s \cup (t \cup r))\}$, by remarks 2.1 and 2.2

   $= \{\phi((s \cup t) \cup r) \mid s \in S, t \in T, r \in R,$ and $\psi((s \cup t) \cup r)\}$, as $\cup$ is associative

   $= \{\phi(\phi(s \cup t) \cup r) \mid s \in S, t \in T, r \in R,$ and $\psi(\phi(s \cup t) \cup r)\}$, by remarks 2.1 and 2.2

   $= \{\phi(st \cup r) \mid st \in S \cap T, r \in R,$ and $\psi(st \cup r)\}$, by definition 12 where st is $\phi(s \cup tr)$

   $= (S \cap T) \cap R$, by definition 12.

3. (a) Null element: $\bot \cap S = S \cap \bot$, from part 1

      $= \{\phi(s \cup t) \mid s \in S, t \in \bot,$ and $\psi(s \cup t)\}$, by definition 12

      $= \{\}$, since $\nexists t \in \bot$

      $= \bot$, by definition 8.

   (b) Uniqueness: Let R be a table such that $R \neq \bot$ and for any other table S, $R \cap S = S \cap R = R$. R cannot be empty as this contradicts $R \neq \bot$. If $R = \top$ then R is a unit element, by part 4 and does not have the null property. Let $r \in R$ and let $X = t$ be an equation in r. If t is a function, then let $X = a$ be an equation in some $s \in S$ such that $a \neq t$. Then $\psi(\{X = t\} \cup \{X = a\})$ is false, and so is $\psi(s \cup r)$ which leaves R empty causing a contradiction. Hence, r can only be a variable-pure substitution. In this case let

$\tau_S \cap \tau_R = \emptyset$. Then $\psi(s \cup r)$ is true for any $s \in S$ and any $r \in R$ and $\phi(s \cup r) = \phi(s) \cup \phi(r) = s \cup r$, by remark 2.3 and the fact that R and S are tables. But $s \cup r$ cannot be in R under the last assumption, which again makes R empty. Hence, R must be $\bot$.

4. (a) Unit element: $\top \cap S = S \cap \top$, from part 1
   $= \{\phi(s \cup t) \mid s \in S, t \in \top, \text{ and } \psi(s \cup t)\}$, by definition 12
   $= \{\phi(s \cup \{\}) \mid s \in S, \text{ and } \psi(s \cup \{\})\}$, by definition 9
   $= \{\phi(s) \mid s \in S, \text{ and } \psi(s)\}$, by definition of $\cup$
   $= \{s \mid s \in S\} = S$, by definition 3 as S is a table.

   (b) Uniqueness: Follows a similar argument to the null case.

□

The fact that $\bot$ is the null element of intersection corresponds to the fact that any query fails to have a successful SLD-derivation if any one of its goals by itself has that property. Similarly, the fact that $\top$ is the unit of intersection corresponds to the fact that inserting anywhere in a query G a successful goal without variables does not affect whether G has successful derivations or its answer substitutions, if any.

"$\cap$" is monotone as shown by the following:

Theorem 4 *For all tables T_1, T_2, S_1 and S_2,*

$$T_1 \subseteq S_1 \text{ and } T_2 \subseteq S_2 \Rightarrow T_1 \cap T_2 \subseteq S_1 \cap S_2.$$

Proof $\eta \in T_1 \cap T_2$

$\Rightarrow \eta = \phi(\theta_1 \cup \theta_2)$ for some $\theta_1 \in T_1$ and some $\theta_2 \in T_2$ such that $\psi(\theta_1 \cup \theta_2)$, by definition 12

$\Rightarrow \eta = \phi(\theta_1 \cup \theta_2)$ for some $\theta_1 \in S_1$ and some $\theta_2 \in S_2$ such that $\psi(\theta_1 \cup \theta_2)$, by assumptions
$\Rightarrow \eta \in S_1 \cap S_2$, by definition 12.
□

The following theorem formalizes Colmerauer's observation. In chapter 3, we show how this gives our framework an advantage over Horn clause form.

Theorem 5 *If P is a program and G_1, G_2, and G are queries such that G consists of all goals in G_1 and G_2, then*

$$(G_1 \texttt{ where } P) \cap (G_2 \texttt{ where } P) = G \texttt{ where } P.$$

Note that, as the order of goals in a query does not matter, the left argument of `where` can be regarded as a set.

Proof $s \in (G_1 \texttt{ where } P) \cap (G_2 \texttt{ where } P)$
$\Leftrightarrow \exists s_1, s_2$ such that $s_1 \in (G_1 \texttt{ where } P)$, $s_2 \in (G_1 \texttt{ where } P)$, $\psi(s_1 \cup s_2)$ and $s = \phi(s_1 \cup s_2)$, by definition 12
$\Leftrightarrow \exists s_1, s_2$ such that there are SLD-refutations with respect to P from G_1 and G_2 with answer substitutions s_1 and s_2 respectively and $\psi(s_1 \cup s_2)$ and $s = \phi(s_1 \cup s_2)$, by definition 7
$\Leftrightarrow G = G_1 \cup G_2$ succeeds with s, by remark 2.2 where $s = \phi(s_1 \cup s_2)$
$\Leftrightarrow s \in (G \texttt{ where } P)$, by definition 7.
□

2.3 Relational application: transforming relations to tables

Although we have a table-producing operator, namely `where`, we still need an operation to get a table out of a relation, rather than from a program by posing a query.

Consider the Prolog query

```
?- qsort([X|Xs],U-W)
```

where `[X|Xs]` is Prolog notation for $cons(X, Xs)$, and `U-W` is infix notation for $-(U, W)$. The answers to this query form a table, with columns headed by X, Xs, U, and W. This table is obtained from the binary relation denoted by "`qsort`" and from the pair of terms (`[X|Xs]`,`U-W`). We regard the table as the result of the *application* operator ":" on a relation and a tuple of terms. We introduce its definition through a heuristic development.

Given an n-ary relation r, a table "most like" r can be easily constructed by adding as headings an arbitrary set of n distinct variables $x_1, \ldots, x_n$. For example, if r_1 is the ternary relation $\{(a, b, c), (a, c, b), (b, a, c)\}$, and X, Y and Z are variables, then table 2.4 is such a table for relation r_1.

X	Y	Z
a	b	c
a	c	b
b	a	c

Table 2.4: The table "most like" relation r_1.

To define such a table, "most like" a relation, we use `where` and an arbitrary predicate symbol, say p, as follows:

$$(\leftarrow p(x_1, \ldots, x_n)) \texttt{ where } \{p(e_1, \ldots, e_n) \mid (e_1, \ldots, e_n) \in r\}.$$

In other words, we make a program P out of the given relation r and the arbitrary predicate symbol p; we include in P all ground facts $p(e_1, \ldots, e_n)$ such that $(e_1, \ldots, e_n) \in r$. Out of this program we get the desired table by choosing for the `where` query a goal of the form $p(x_1, \ldots, x_n)$.

The obvious generalization of allowing any terms $t_1, \ldots, t_n$ instead of the distinct variables suggests

$$r : (t_1, \ldots, t_n) \stackrel{\text{def}}{=} (\leftarrow p(t_1, \ldots, t_n)) \texttt{ where } \{p(e_1, \ldots, e_n) \mid (e_1, \ldots, e_n) \in r\}.$$

Though a correct definition, the auxiliary predicate symbol "p" is undesirable. Notice that the answer substitutions forming a table by this definition are the results of successful unifications of the tuple of terms in the query with each tuple of r; i.e. all the solved forms of $\{t_1 = e_1, \ldots, t_n = e_n\}$ where $(e_1, \ldots, e_n) \in r$. Using the ϕ and ψ operators, we can get an equivalent definition that needs no auxiliary predicate symbol.

Definition 13 *The application operation (denoted ":") of an n-ary relation r to a tuple* $(t_1, \ldots, t_n)$ *of terms is*

$r : (t_1, \ldots, t_n) \stackrel{\text{def}}{=}$

$$\{\phi(\{t_1 = e_1, \ldots, t_n = e_n\}) \mid (e_1, \ldots, e_n) \in r \textit{ and } \psi(\{t_1 = e_1, \ldots, t_n = e_n\})\}.$$

For example, $T_1 = r_1 : (X, Y, Z)$ is table 2.4, while $T_2 = r_1 : (a, Y, Z)$ is table 2.5. Notice that the arity of the resulting table is determined by the tuple argument of ":"; it is equal to the number of distinct variables in the tuple.

The following remarks, about how ":" yields $\bot$ and $\top$, follow from definition 13. For any terms $t_1, \ldots, t_n$,

$$\{\} : (t_1, \ldots, t_n) = \bot. \tag{2.4}$$

Y	Z
b	c
c	b

Table 2.5: $T_2 = r_1 : (a, Y, Z)$.

For any variable-free terms $t_1, \ldots, t_n$, and a relation r,

$$r : (t_1, \ldots, t_n) = \begin{cases} \top & \text{if } (t_1, \ldots, t_n) \in r \\ \bot & \text{otherwise.} \end{cases} \tag{2.5}$$

We also show that ":" is monotone as follows:

Theorem 6 *For any k-ary relations r and p, and any terms $t_1, \ldots, t_k$,*

$$r \subseteq p \Rightarrow r : (t_1, \ldots, t_k) \subseteq p : (t_1, \ldots, t_k).$$

Proof $s \in r : (t_1, \ldots, t_k)$
$\Rightarrow s = \phi(\{t_1 = s_1, \ldots, t_k = s_k\})$ for some tuple $(s_1, \ldots, s_k) \in r$ such that $\psi(\{t_1 = s_1, \ldots, t_k = s_k\})$, by definition 13
$\Rightarrow s = \phi(\{t_1 = s_1, \ldots, t_k = s_k\})$ for some tuple $(s_1, \ldots, s_k) \in p$ such that $\psi(\{t_1 = s_1, \ldots, t_k = s_k\})$, by assumptions
$\Rightarrow s \in p : (t_1, \ldots, t_k)$, by definition 13.
□

2.4 Relational projection: from tables to relations

In our intuitive view of Horn clauses, the last step in concluding the relation defined by a clause is *projection*. This step projects the intersection of the

applications of the conditions onto the variables in the conclusion. As we have formalized those applications as tables, and their intersection yields a table, this operation should get a relation from a table.

This operation, however, should not merely discard the table's "headings," as suggested in [21]. One reason is, as mentioned before, that *tables contain variables, whereas the tuples of a relation are ground.* Moreover, we need to take the variables in the head of a clause into account. We follow a heuristic development to define this operation properly.

Let T be a table with an n-ary set $\{x_1, \ldots, x_n\}$ of distinct variables as headings, on the left-hand sides of equations, and with ground terms as entries, on the right-hand sides of equations. Consider $p = \{(x_1\theta, \ldots, x_n\theta) \mid \theta \in T\}$. This is the suggested approximation, that is, a relation resulting from only discarding the headings. For example, for table T_1 above, $\{(X\theta, Y\theta, Z\theta) \mid \theta \in T_1\}$ gives $\{(a, b, c), (a, c, b), (b, a, c)\}$; i.e. r_1 above. To generalize, we lift the variable and the arity restrictions from the tuple, and we allow the table to include variables as entries.

First let $\{j_1, \ldots, j_k\}$ be a subset of n. Then, $p' = \{(x_{j_1}\theta, \ldots, x_{j_k}\theta) \mid \theta \in T\}$ is a projection of p on the sequence $j_1, \ldots, j_k$ of columns. Hence, projection is a way of getting a relation from a table. To generalize, however, we allow the tuple to be of an arbitrary arity, say m, so that m could be even greater than n, the arity of T, and the variables could be other than those in T's heading.

Next, allow any terms $t_1, \ldots, t_m$ in the tuple instead of the variables $x_1, \ldots, x_m$ and consider $p'' = \{(t_1\theta, \ldots, t_m\theta) \mid \theta \in T\}$. For an arbitrary table T, any $\theta \in T$ may contain variables as entries. Therefore, the result must be made ground in order to obtain a relation. Lifting the arity restriction of the

tuple allows the same problem to arise. To facilitate both generalizations, we introduce the auxiliary "ξ" operator.

Informally, given an arbitrary expression x, $\xi(x)$ is the set of variable-free instances of x, with respect to the Herbrand universe. Formally, we define it inductively as follows.

Definition 14 *The ground instantiation (or simply grounding) operator* ξ

- *For a constant* c, $\xi(c) \stackrel{\text{def}}{=} \{c\}$.
- *For a variable* v, $\xi(v) \stackrel{\text{def}}{=} U_H$; *i.e. the Herbrand universe.*
- *For an n-ary function* $f(t_1, \ldots, t_n)$,
  $\xi(f(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} \{f(g_1, \ldots, g_n) \mid (g_1, \ldots, g_n) \in \xi((t_1, \ldots, t_n))\}$.
- *For an n-tuple* $(t_1, \ldots, t_n)$ *of terms,*
  $\xi((t_1, \ldots, t_n)) \stackrel{\text{def}}{=} \{(t_1, \ldots, t_n)\phi(\{t_1 = g_1, \ldots, t_n = g_n\}) \mid g_i \in \xi(t_i), i \in n, \psi(\{t_1 = g_1, \ldots, t_n = g_n\})\}$.
- *For an n-ary set* $\{x_1 = t_1, \ldots, x_n = t_n\}$ *of equations in solved form,*
  $\xi(\{x_1 = t_1, \ldots, x_n = t_n\}) \stackrel{\text{def}}{=} \{\{x_1 = g_1, \ldots, x_n = g_n\} \mid (g_1, \ldots, g_n) \in \xi((t_1, \ldots, t_n))\}$.
- *For a set of tuples or a table* T, $\xi(T) \stackrel{\text{def}}{=} \bigcup_{t \in T} \xi(t)$.

Thus, a ground table T, one with no variable as entry, can be defined by $\xi(T) = T$. With this tool, we can now generalize the definition of projection as follows:

Definition 15 *The projection operation (denoted "/") of a table T on a tuple $(t_1, \ldots, t_n)$ of terms is*

$$(t_1, \ldots, t_n)/T \stackrel{\text{def}}{=} \xi(\{(t_1\theta, \ldots, t_n\theta) \mid \theta \in T\}).$$

For example, assuming $U_H = \{a, b, c\}$,

$$(X, Y, Z)/T_1 = r_1,$$

$$(a, Y, Z)/T_1 = \{(a, b, c), (a, c, b), (a, a, c)\},$$

$$(X, Y, Z)/T_2 = \{(a, b, c), (b, b, c), (c, b, c), (a, c, b), (b, c, b), (c, c, b)\}, \text{ and}$$

$$(X, Y, X)/T_2 = \{(a, b, a), (b, b, b), (c, b, c), (a, c, a), (b, c, b), (c, c, c)\}.$$

Those familiar with the `setof` primitive of Prolog will see a similarity. The tuple operand of "/" takes the place of the template term in `setof` and the table operand takes the place of the query in `setof`, while the result corresponds to the answer list in `setof`. The difference is that `setof`, being a predicate, requires a parameter for its result, while "/" does not.

For any tuple $(t_1, \ldots, t_n)$ of terms,

$$(t_1, \ldots, t_n)/\bot = \{\}, \text{ and} \tag{2.6}$$

$$(t_1, \ldots, t_n)/\top = \xi((t_1, \ldots, t_n)). \tag{2.7}$$

If the terms $t_1, \ldots, t_n$ are variable-free, then for any table $T \neq \bot$,

$$(t_1, \ldots, t_n)/T = \{(t_1, \ldots, t_n)\}. \tag{2.8}$$

"/" is monotone as shown by the following:

Theorem 7 *If T and S are tables and $t_1, \ldots, t_n$ are terms, then*

$$T \subseteq S \Rightarrow (t_1, \ldots, t_n)/T \subseteq (t_1, \ldots, t_n)/S.$$

Proof $e \in (t_1, \ldots, t_n)/T$

$\Rightarrow e \in \xi((t_1\theta, \ldots, t_n\theta))$ for some substitution $\theta \in T$, by definition 15

$\Rightarrow e \in \xi((t_1\theta, \ldots, t_n\theta))$ for some substitution $\theta \in S$, by assumptions

$\Rightarrow e \in (t_1, \ldots, t_n)/S$, by definition 15.

□

2.5 Invertibility of "/" and ":"

Now that we have operations from tables to relations and vice versa, one may wonder whether these are each other's inverses. The short answer is, in general, "no", because

$$((t_1, \ldots, t_n)/T) : (t_1, \ldots, t_n)$$

is not always the table T. Take, for example, the case that $t_1, \ldots, t_n$ are ground. Then, by remarks 2.8 and 2.5, the above expression is the unit table $\top$ whenever T is not the null table $\bot$. But the absence of variables in $t_1, \ldots, t_n$ is a rather pathological case. When we add restrictions, we can say that, in a sense, ":" and "/" are each other's inverses, as shown by the following theorems.

Theorem 8 *If T is a table and $t_1, \ldots, t_n$ are terms in which all the variables, and no other ones, in τ_T occur, then*

$$((t_1, \ldots, t_n)/T) : (t_1, \ldots, t_n) = \xi(T).$$

Proof $((t_1, \ldots, t_n)/T) : (t_1, \ldots, t_n)$

$= \xi(\{(t_1\theta, \ldots, t_n\theta) \mid \theta \in T\}) : (t_1, \ldots, t_n)$, by definition 15

$= \bigcup_{\theta \in T} \xi((t_1\theta, \ldots, t_n\theta)) : (t_1, \ldots, t_n)$, by definition 14

$= \{\phi(\{t_1 = e_1, \ldots, t_n = e_n\}) \mid (e_1, \ldots, e_n) \in \bigcup_{\theta \in T} \xi((t_1\theta, \ldots, t_n\theta))$ and $\psi(\{t_1 = e_1, \ldots, t_n = e_n\})\}$, by definition 13

$= \{\theta' \mid \theta' \in \xi(\theta), \theta \in T\}$, as shown below

$= \bigcup_{\theta \in T} \xi(\theta) = \xi(T)$, by definition 14.

The justification of the main step is as follows. Let t be $(t_1, \ldots, t_n)$ and let e be the arbitrary element $(e_1, \ldots, e_n)$ in the left-hand side of the equality. $e = (t\theta)\eta$ for some ground substitution η. Hence, by the assumptions, the variables in τ_η are those in the right-hand sides of θ which are distinct from those in τ_θ, as θ is in solved form being an element of a table. So, by the definition of composition of substitutions, $\theta\eta = \theta' \cup \eta$ where $\theta' \in \xi(\theta)$ and with no variable in common between θ' and η. Thus, by proposition 1, $(t\theta)\eta = t(\theta\eta) = t(\theta' \cup \eta) = t\theta'$, since the variables in t are exactly those in $\tau_{\theta'}$. Hence, by the unification algorithm, t and $e = t\theta'$ unify with θ' as mgu. □

As a special case, we get the following:

Corollary 1 *If T is a table and $x_1, \ldots, x_n$ are distinct variables such that $\{x_1, \ldots, x_n\} \subseteq \tau_T$, then*

$$((x_1, \ldots, x_n)/T) : (x_1, \ldots, x_n) = \xi(T)[\{x_1, \ldots, x_n\}],$$

that is, the restriction of $\xi(T)$ to the subset $\{x_1, \ldots, x_n\}$ of its left-hand side variables[5].

[5] Using the notation of set theory, this is $\pi_{\{x_1, \ldots, x_n\}}\xi(T)$.

For an inverse in the other direction, compare the n-ary relation r with

$$(t_1, \ldots, t_n)/(r : (t_1, \ldots, t_n)).$$

That this expression does not always equal r is shown by

$$(c, d)/(\{(a, b)\} : (c, d)) = \{\},$$

where a, b, c and d are constants. This example suggests:

Theorem 9 *If r is an n-ary relation and $t_1, \ldots, t_n$, are terms, then*

$$(t_1, \ldots, t_n)/(r : (t_1, \ldots, t_n)) \subseteq r.$$

Proof Suppose $(e_1, \ldots, e_n)$ is in the left hand side. Then, by definition 15, $\exists \eta \in (r : (t_1, \ldots, t_n))$ and a ground substitution θ such that $e_i = (t_i \eta)\theta$ for $i \in n$. Moreover, by definition 13, $\eta = \phi(\{t_1 = f_1, \ldots, t_n = f_n\})$ for a tuple $(f_1, \ldots, f_n) \in r$. Hence, $f_i = t_i \eta$ for $i \in n$, so that $t_i \eta$ is ground. Hence, $e_i = (t_i \eta)\theta = t_i \eta = f_i$ for $i \in n$. We conclude that $(e_1, \ldots, e_n) \in r$.
□

However, by strengthening the restrictions, we can have equality instead of inclusion, as shown in the following theorem.

Theorem 10 *If r is an n-ary relation and $x_1, \ldots, x_n$ are distinct variables, then*

$$(x_1, \ldots, x_n)/(r : (x_1, \ldots, x_n)) = r.$$

Proof One part of the theorem is already true by the previous theorem, so that we only need to prove that $(x_1, \ldots, x_n)/(r : (x_1, \ldots, x_n)) \supseteq r$.

Now, $(e_1, \ldots, e_n) \in r$
$\Rightarrow \{x_1 = e_1, \ldots, x_n = e_n\} \in r : (x_1, \ldots, x_n)$, by definition 13 since $\{x_1 =$

$e_1, \ldots, x_n = e_n\}$ is in solved form under the assumptions, and with observing that

$$\xi((x_1, \ldots, x_n)\{x_1 = e_1, \ldots, x_n = e_n\}) = \xi((e_1, \ldots, e_n)) = \{(e_1, \ldots, e_n)\}$$

$\Rightarrow (e_1, \ldots, e_n) \in (x_1, \ldots, x_n)/(r : (x_1, \ldots, x_n))$, by definition 15.

□

Chapter 3

A basis for relational programming

Our goal in this thesis is to provide an applicative framework for relational programming, especially as an extension for logic programming. In this chapter, we show how the formalism we developed is such a framework. We show how our applicative framework can accommodate definite Horn clauses. But this is only justified if the expected advantages of such a framework, namely modularity, local scoping, and higher-order constructs, can also be accomplished. Hence, we demonstrate as well that our framework has these advantages.

As Joseph Stoy puts it in [20],

> many treatments [of the semantics of programming languages] split programming languages up into two parts: a descriptive part (sometimes called the applicative part) and an imperative part; these are also sometimes known as the expression part and the command part. One possible plan [for studying the semantics of a language] is to study the descriptive part of a language first,

> so that the ideas there may be thoroughly understood before the extra complications of commands are introduced.

In that sense, our framework constitutes an expression part that can be supplemented by a chosen set of constructs to form a programming language, an applicative relational one.

3.1 Applicative and Horn-clause definitions

The operators of our framework provide for a simple translation scheme from definite Horn clauses to applicative form. Therefore, we demonstrate the translation by examples and leave an explicit description of it to the results in this section.

The principle of the translation is that each definite Horn clause states that *the relation denoted by the predicate symbol in the conclusion includes the projection (on the tuple of the terms in the conclusion) of the intersection of the tables denoted by the conditions.* Our operations are general enough to translate the definite Horn clauses of pure Prolog.

This shows that a language based on our applicative framework would be at least as expressive as the Horn-clause form. But how do the inclusions uniquely define their respective relations, and how do our operators and values constitute expressions with relations as their values? The following development arrives at that.

We assume, in the applicative language, a method for naming logic programs, that is sets of definite Horn clauses. This way, the second argument of `where` can be a name rather than a set expression explicitly listing the constituent clauses.

Consider the logic program P_{num} defining the number relation *num*, in terms of the successor function, as follows:

```
num(0).
num(s(X)) :- num(X).
```

Although it defines the relation *num*, P_{num} does not contain any expression whose value is the relation *num*. As a first step towards an expression for *num*, consider the following applicative expression, call it P_{num_1}:

$$(X)/((\leftarrow \mathrm{num}(X)) \texttt{ where } P_{\mathrm{num}}).$$

Because it incorporates P_{num}, P_{num_1} does yield the number relation as its value. As the number relation is now an individual, we can denote it by a constant term. Moreover, the individual can be the value of a *first-order variable*, say *num*[1], so that we can write:

$$num = (X)/((\leftarrow \mathrm{num}(X)) \texttt{ where } P_{\mathrm{num}}).$$

This equality holds because its right-hand side is defined mainly in terms of SLD-resolution, which is proved to be both complete and correct; the projection serves to transform the table into a relation. The improvement is that we now have an expression P_{num_1} and a first-order variable *num* each of which has the number relation as its value.

Moreover, such an expression shows that an applicative language can have Prolog as sublanguage. This is achieved via the `where` operator, whose

[1] We use strings beginning with lower-case letters for relational variables in order to differentiate them from logic variable while retaining their similarity to predicate symbols. They are still distinguished from both predicate symbols and functors by the way in which they are used, just as with the latter two. Hence, *num* as a relational variable should not be confused with the predicate symbol num.

second operand is a program that can be entirely in Prolog, or only partly Prolog because of embedded `where`'s.

Once we have *relational* variables, we can show how the applicative framework can by itself define the number relation, and in a way that is intertranslatable with Horn clauses. Consider the following applicative expression, call it P'_{num}:

$$num \supseteq ((0)/\top) \cup ((s(X))/(num : (X))),$$

where num is a relational variable. P'_{num} states an inclusion between relations and may or may not be satisfied, depending on the value of num. P'_{num}, however, can be viewed as a recursive program scheme in the sense of de Bakker and de Roever [10]. Hence, a minimization operator, like μ of [10], can be applied to the recursive scheme P'_{num} with respect to num giving a least relation for num that satisfies the inclusion. By letting the inclusion be a definition for this least relation, P'_{num} becomes a relation-valued expression.

We already introduced a minimization operator, namely ν. We used it as a unary operator to get the least Herbrand model of a program. We need to introduce another distinct usage of ν as a *binary* operator. It takes, in addition to the program argument, a *variable* denoting a relation in the program operand; e.g. $\nu p.P$ where p is a relational variable and P is a program. The result is then the least relation as value for that variable in νP.

The least Herbrand model of a program P, νP, is equal to $T_P^{\omega}(\emptyset)$, where T is the so-called *immediate-consequence* transformation of [23]. On the other hand, the relations determined by an interpretation (and hence by a model too) can be obtained by the operator $\diamond$ used in [2].

Definition 16 *For an interpretation I and an n-ary predicate symbol p,*

$$I \diamond p = \{(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n) \in I\}.$$

However, $\nu P \diamond p$, though relation-valued, cannot be a definition for $\nu p.P$ where P is a definite program, since p can only be a predicate symbol there; only in a higher-order formalism can we have a *predicate variable.*

Our framework, as we have shown, has relational variables and expressions. As such, we can have $\nu p.P'$ where P' is an applicative relational program and p a relational variable. Of course, for a specific programming language based on our framework we need to define a T transformation in terms of our operators and the additional constructs of that language. However, our goal here is to show that our applicative formalism is equivalent to the declarative Horn-clause formalism. In this scope, we can assume the existence of a minimization operator, as shown below.

Notice first that P'_{num} above can be obtained from P_{num} by a simple translation based on our intuitive view of Horn clauses: the relational operations in Horn clauses are made explicit, using our operators, and the procedure for *num* then becomes a relational inclusion whose left-hand side is *num* as a relational variable, and the right-hand side is a union of the resulting projections. In fact, the translation can be made simpler using the alternative form:

$$\begin{aligned} num &\supseteq (0)/\top \\ num &\supseteq (s(X))/(num : (X)), \end{aligned}$$

in which the union is made implicit.

Clearly, this suffices to translate any Horn-clause program into applicative form, and, using a reverse scheme, vice versa. Hence, for the class of relational programs determined by the translation into Horn-clause form, we can be sure of the existence of a minimization operator. In other words, our strategy is to translate a relational program to a logic program and to define the meaning of the relational program as the meaning of the corresponding logic program. This is shown by figure 3.1.

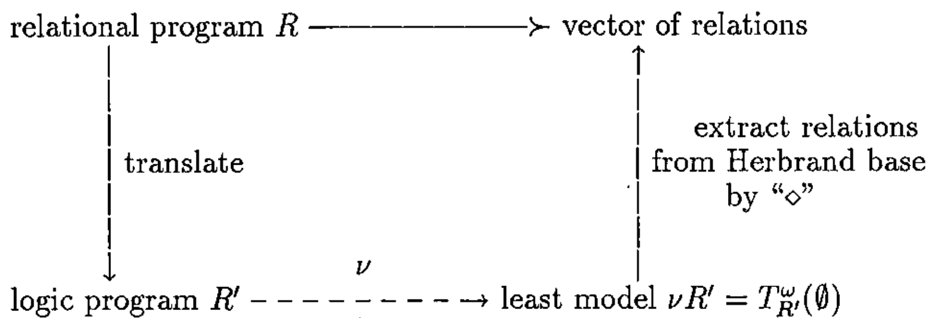


Figure 3.1: The meaning of a relational program.

First we define a transformation scheme to map any relational program into the corresponding logic program; see appendix B. Then we use the fix-point semantics of logic programming to prove, for the latter, the existence of the least (Herbrand) model, and thus the ν operator. Finally, we can extract any defined relation in the original relational program using the $\diamond$ operator. Hence, the value defined for a relational variable p in a relational program R is $\nu R' \diamond \mathrm{p}$, where R' is the logic program resulting from the translation of (inclusions of) R to clauses and p is a predicate symbol in R' denoting the relation as value of p.

In this way, we can have $\nu num.P'_{num}$ as a least relation as value for num that satisfies the inclusion P'_{num}; i.e. $\nu num.P'_{num} = \nu P_{\mathbf{num}} \diamond \mathrm{num}$. The inclusion then serves as definition of this least relation. Thus P'_{num}, an applicative

expression, defines the number relation uniquely (and recursively) while allowing the first-order variable *num* to denote its value.

The following theorems justify our approach. The first determines a class of relational programs that can translate any Horn-clause program, but restricts the translation of conditions to `where` expressions.

Theorem 11 *For all atoms $p(t_1,\ldots,t_n), B_1,\ldots,B_m$ and a program P,*

$$p(t_1,\ldots,t_n) \leftarrow B_1,\ldots,B_m \textit{ is true in } \nu P \Leftrightarrow$$

$$\nu P \diamond p \supseteq (t_1,\ldots,t_n)/((\leftarrow B_1 \texttt{ where } P) \cap \cdots \cap (\leftarrow B_m \texttt{ where } P)).$$

Proof $p(t_1,\ldots,t_n) \leftarrow B_1,\ldots,B_m$ is true in νP
$\Leftrightarrow p(t_1,\ldots,t_n) \leftarrow ((\leftarrow B_1,\ldots,B_m) \texttt{ where } P)$ is true in νP, by theorem 2
$\Leftrightarrow \xi(\{p(t_1\theta,\ldots,t_n\theta) \mid \theta \in ((\leftarrow B_1,\ldots,B_m) \texttt{ where } P)\}) \subseteq \nu P$, by definition 10
$\Leftrightarrow \xi(\{(t_1\theta,\ldots,t_n\theta) \mid \theta \in ((\leftarrow B_1,\ldots,B_m) \texttt{ where } P)\}) \subseteq \nu P \diamond p$, by definition 16
$\Leftrightarrow (t_1,\ldots,t_n)/((\leftarrow B_1,\ldots,B_m) \texttt{ where } P) \subseteq \nu P \diamond p$, by definition 15
$\Leftrightarrow (t_1,\ldots,t_n)/((\leftarrow B_1 \texttt{ where } P) \cap \cdots \cap (\leftarrow B_m \texttt{ where } P)) \subseteq \nu P \diamond p$, by theorem 5.
□

Corollary 2 *For any atom $p(t_1,\ldots,t_n)$ and any program P,*

$$p(t_1,\ldots,t_n) \leftarrow \textit{ is true in } \nu P \Leftrightarrow \nu P \diamond p \supseteq (t_1,\ldots,t_n)/\top.$$

Notice that each goal $B_i = \mathrm{r}(t_1,\ldots,t_k)$ is local to its `where` expression so that the resulting table can be replaced by any other table with the same value, say $r : (t_1,\ldots,t_k)$, where r is a relational variable whose value is

$\nu P \diamond \mathrm{r}$. Hence, in the case where the terms are distinct variables, we can prove the equivalence of our applicative translation with the Horn clause form, as follows.

Theorem 12 *For a program P, an n-ary predicate symbol* p*, and distinct variables $x_1, \ldots, x_n$,*

$$\xi(\leftarrow \mathrm{p}(x_1, \ldots, x_n) \texttt{ where } P) = p : (x_1, \ldots, x_n)$$

where p is a relational variable whose value is the least relation determined by P for p*; i.e. $\nu P \diamond \mathrm{p}$.*

Proof $\xi(\leftarrow \mathrm{p}(x_1, \ldots, x_n) \texttt{ where } P)$
$= (x_1, \ldots, x_n)/(\leftarrow \mathrm{p}(x_1, \ldots, x_n) \texttt{ where } P) : (x_1, \ldots, x_n)$, by theorem 8
$= \nu P \diamond \mathrm{p} : (x_1, \ldots, x_n)$, by definitions 7, 15, 2 and 16 and by lemmas 1 and 2
$= p : (x_1, \ldots, x_n)$, where p is a relational variable denoting relation $\nu P \diamond \mathrm{p}$.
□

As our translations are inclusions, we can remove the variable restriction by the following

Theorem 13 *For any atom $\mathrm{p}(t_1, \ldots, t_n)$ and a program P,*

$$\xi(\leftarrow \mathrm{p}(t_1, \ldots, t_n) \texttt{ where } P) \subseteq p : (t_1, \ldots, t_n)$$

where p is a relational variable whose value is $\nu P \diamond \mathrm{p}$.

Proof $(t_1, \ldots, t_n)/(\leftarrow \mathrm{p}(t_1, \ldots, t_n) \texttt{ where } P) \subseteq \nu P \diamond \mathrm{p}$, by definitions 7, 15, 2 and 16 and by lemma 1
$\Rightarrow (t_1, \ldots, t_n)/(\leftarrow \mathrm{p}(t_1, \ldots, t_n) \texttt{ where } P) : (t_1, \ldots, t_n) \subseteq (\nu P \diamond \mathrm{p}) : (t_1, \ldots, t_n)$, by theorem 6

$\Rightarrow \xi(\leftarrow \mathrm{p}(t_1, \ldots, t_n)$ `where` $P) \subseteq (\nu P \diamond \mathrm{p}) : (t_1, \ldots, t_n)$, by theorem 8

$\Rightarrow \xi(\leftarrow \mathrm{p}(t_1, \ldots, t_n)$ `where` $P) \subseteq p : (t_1, \ldots, t_n)$, where p is a relational variable denoting $\nu P \diamond \mathrm{p}$.

□

This completes the proof of the existence of the minimization operator for our applicative translations of Horn clauses, by theorems 4 and 7. Only a limited class of relational programs escapes this proof, ones that contain a non-ground table. This restriction remains because, in general, for tables T_1 and T_2

$$\xi(T_1 \cap T_2) \neq \xi(T_1) \cap \xi(T_2).$$

The examples in the following section are all in the proved class.

3.2 Advantages of applicative expressions

The usefulness of applicative expressions in programming is the subject of this section. We demonstrate how to provide for local scoping, modularity, and relations as first-class objects.

3.2.1 Modular logic programming

When we write a Horn clause

$$A \leftarrow B_1, \ldots, B_n$$

it is assumed understood with respect to which logic program the condition part is to be evaluated. Of course it is taken for granted that all goals are evaluated with respect to the same program. While such default assumptions are often convenient, modular programming requires alternatives, one of which is suggested by tables as conditions of the condition part.

Let us for the time being continue to assume that all goals in a condition part are evaluated with respect to the same program. The concept of table, and an expression for it, make it possible to explicitly specify that program. In the first place, the `where` operator provides a scope for the condition part of a clause making the goals local to the `where` expression. This allows different clauses in the same program to have their condition part specified in different programs, as in

$$A \leftarrow ((\leftarrow B_1, B_2, B_3) \texttt{ where } P)$$
$$C \leftarrow ((\leftarrow D_1, D_2) \texttt{ where } Q).$$

Intersection of tables further facilitates writing the condition part of a Horn clause as a table-valued expression. It allows different goals in the same condition part to be evaluated with respect to different programs, as in

$$E \leftarrow (((\leftarrow F_1, F_2) \texttt{ where } P) \cap ((\leftarrow G) \texttt{ where } Q)).$$

In this way, the combination of `where` and $\cap$ can provide locality for each goal in each clause of a program.

This facility extends naturally to queries as they have the same type as condition parts. This gives the ability to compose more complex queries than would be possible using Horn clauses, e.g. as complex as

$$(G_1 \texttt{ where } P_1) \cap (G_2 \texttt{ where } P_2) \cap \cdots \cap (G_n \texttt{ where } P_n)$$

for $n = 1, 2, \ldots$ An example of a query of this complexity is the operation of the TuplePipes logic programming system [6]. In this system, the user is typically in the process of adding a goal to an open-ended query. The user sees in tabular form the answer substitutions to the query so far. If

this table is satisfactory, the user halts the query. Otherwise, she[2] adds a goal, resulting in a table consisting of the answer substitutions to the newly incremented query. Hence, the above query is appropriate to describe the stream of tables that the user sees successively. In TupiLog [6], the existing implementation of TuplePipes, this is restricted to $P_1 \subseteq \cdots \subseteq P_n$.

3.2.2 Modular relational programming: exporting and importing relations

As an example of the program-structuring mechanism that we need in relational programming, consider

```
Program Part:
partition([],[],[],Y).
partition([X|Xs],[X|Ls],Bs,Y)<-order(X,Y),partition(Xs,Ls,Bs,Y).
partition([X|Xs],Ls,[X|Bs],Y)<-order(Y,X),partition(Xs,Ls,Bs,Y).
```

It defines the 4-ary relation *partition* of a list according to an item into two lists; i.e. the second argument contains all the items of the first argument that are not greater in order than the last argument and the third one contains all the greater items, or vice versa. The order is determined by the predicate symbol `order` whose definition is lacking. Suppose we want it to be `leq`, less or equal, as defined by

```
Program Lgeq:
  leq(0,X) <- n(X).
  leq(s(X),s(Y)) <- geq(Y,X).
```

[2]or "he"; as a general rule, if no gender can be deduced from the context, none is implied.

```
geq(X,0) <- n(X).
geq(X,Y) <- leq(Y,X).
```

where n stands for the number relation, and s is the successor function.

What we need when building up large programs from small modules is a mechanism for *importing* into program Part a relation to serve as value for the undefined predicate symbol order. And we need to be able to *export* from Lgeq either one of the two relations defined there.

The λ-calculus takes care of the importation requirement. Suppose r is a binary relation. Then the result of importing it into Part as the value of the binary predicate symbol order is $(\lambda order.\texttt{Part})r$. However, order in Part is a predicate symbol. To make it a variable, a higher-order formalism is needed. Applicative definitions allow us to rewrite Part in such a way that it contains order as a first-order, relation-valued variable, namely into *Program Part′*:

$$
\begin{aligned}
partition &\supseteq ([\,],[\,],[\,],Y)/\top. \\
partition &\supseteq ([X|Xs],[X|Ls],Bs,Y)/ \\
&\quad order:(X,Y)\cap partition:(Xs,Ls,Bs,Y). \\
partition &\supseteq ([X|Xs],Ls,[X|Bs],Y)/ \\
&\quad order:(Y,X)\cap partition:(Xs,Ls,Bs,Y).
\end{aligned}
$$

Notice that the individual (or logical) variables are local to their respective inclusions. Hence, in the alternate form using union, these variables do not require renaming.

Thus our applicative framework enables exploiting λ-abstraction and application as an importation mechanism.

What remains is to enable exporting a relation from `Lgeq`. As it stands, `Lgeq` defines both *leq* and *geq*, greater or equal, between numbers such that there is no way to tell which is the "main" relation. Depending on the occasion where the module $Part'$ is used, one may want one or the other of the two relations defined in `Lgeq`. For example, to make the following `Qsrt` program produce a list in an ascending order we need *leq* as the order in $Part'$.

```
Program Qsrt:
qsort([],U-U).
qsort([X|Xs],U-W) <- partition(Xs,Y1,Y2,X),
                     qsort(Y1,U-[X|V]), qsort(Y2,V-W).
```

The binary "ν" operator takes care of the exportation requirement. As defined in the preceding section, the binary "ν" operator allows us to select one of the relations defined in `Lgeq`. For example, $\nu leq.Lgeq'$, where *leq* is a relational variable and $Lgeq'$ is the relational version of `Lgeq`, is a relation-valued expression that can replace the relation denoted by *order* as, for example, the operator argument of relational application.

Once $Part'$ and $Lgeq'$ have been defined, we need not do anything else than to glue them together without referring to their internals. This is allowed by the combination of "λ" and "ν". In the above example, the result of gluing is denoted by the expression $(\lambda order.Part')\nu leq.Lgeq'$. In the same way, the relation n, defining the type of items, can be defined. We can import it from relational program P'_{num}, as in $(\lambda n.Lgeq')\nu num.P'_{num}$, or from definite program P_{num} using `where`, as in

$$(\lambda n.Lgeq')((X)/(\leftarrow \mathrm{num}(X)) \texttt{ where } P_{\mathrm{num}}).$$

Thus we can have the applicative expressions

$$R_X = (\lambda order.Part')\nu leq.((\lambda n.Lgeq')\nu num.P'_{num}),$$

and, assuming $Qsrt'$ to be the relational version of `Qsrt`,

$$(\lambda partition.Qsrt')\nu partition.R_X.$$

The latter expression gives the desired list of numbers in an ascending order. In this way, any number of levels of locality can be built, just as in applicative functional programming.

Moreover, this provides for *genericity*, not just modularity. For example, consider the following[3]:

```
Program Cart:
ordered( pt(X,Y), pt(V,W) ) :- ord1(X,V), ord2(Y,W).
```

`Cart` (short for *Cartesian*) can be used to define the order between points. Consider $Cart'$, the relational version of `Cart`, defined as

$$ordered \supseteq (pt(X,Y), pt(V,W))/(ord1 : (X,V) \cap ord2 : (Y,W)).$$

Now,

$$R_C = (\lambda ord1.\lambda ord2.Cart')(\nu leq.Lgeq')(\nu geq.Lgeq')$$

defines an ordering of two points such that their first coordinates are in ascending order while their second coordinates are in descending order. Hence,

$$(\lambda order.Part')\nu ordered.R_C$$

[3]This is a variant of an example in [17] on the genericity of class templates, an object-oriented extension of logic programming.

imports this ordering into $Part'$ to be used as the order in one partition and its opposite in the other partition, while making its list's items of the type of points. All this is done independently of $Cart'$, in R_C, and of $Part'$, in the latter example, without modifying them in any way. Thus, $Part'$ and $Cart'$ are both generic modules, and the latter is also a composite one.

In this way, our framework enables incorporating the functional operators of the λ-calculus with the binary minimization operator ν, for relational programs, and with its own operators, for definite programs. This provides for an export/import mechanism for relations between programs, thus allowing generic modules.

3.2.3 Advanced query language

We can enhance the query language using applicative expressions, without the need to alter the program in any way. One such enhancement is the ability to project the answer to the query on an arbitrary tuple. For example, if a program P defines only the `father` and `mother` relations, then a query like

$$parents \stackrel{\text{def}}{=} (Ss, F, M)/((father : (F, Ss)) \cap (mother : (M, Ss)))$$

is possible for a logic program P only by defining an auxiliary predicate for the desired projection. This requires changing P, a change that is only effective for this form of projected tuple. In a sense, our framework allows arbitrary templates for providing different views on the answers of the query.

Another possible enhancement is the ability to get the disjunction of the answers to the goals in the query instead of only their conjunction. This is allowed by taking the union of the goals in the query, since each goal represents a table which in turn is a set. For example, in program P above,

the query on the individuals each of which is either a father or a child of a female, say *mary*, is expressible in our framework as

$$((\leftarrow father(S, mary)) \texttt{ where } P) \cup ((\leftarrow mother(mary, S)) \texttt{ where } P).$$

Here, the global aspect of the logical variable S is used. In logic programming, the same answer, for a query with the same meaning, can only be achieved via an auxiliary predicate; taking the conjunction gives a different meaning, one which should fail in a realistic example.

Notice that the first technique can be exploited here to get a result that explains itself, as in

$$fatherOrChild \stackrel{\text{def}}{=} (S, mary)/((father : (S, mary)) \cup (mother : (mary, S))).$$

In other words, the definition of the relations in a logic program implicitly restricts the templates that the user may use for querying that program, namely to only a conjunction of the relations defined in the program. The applicative framework, however, allows arbitrary relational expressions to form a query.

Chapter 4

Related work

Although we do not formalize our tables and relations as an algebra, our work is related to such formalizations; for example, the work of Peirce and of Schroeder in the 19th century. However, Tarski was the first to provide, in the 1930's, an algebra adequate to serve as basis for a semantics for full first-order predicate logic, namely cylindric set algebra [12].

In [10], de Bakker and de Roever combined Tarski's axioms for a relation algebra, restricted to binary relations, with an induction rule for recursion, due to Scott, into a calculus for recursive program schemes. In their work, they elucidated the mechanisms of defining binary relations, especially the use of the minimization operator. As we observed in section 3.1, our applicative framework facilitates extending their definition techniques.

There are important relations to the area of relational database theory. Codd's relational calculus is declarative, whereas his relational algebra is applicative. Both are improved upon by the earlier work by Tarski, as it has been shown in [13]. Our framework can accommodate Codd's relational operators, as we show in appendix B.

Deductive databases, however, have Datalog, a subset of Prolog, as query

language. Hence, our framework has a simple intertranslation with Datalog, shares with it the same inference machine, and can even accommodate Datalog as a sublanguage.

In particular, the class of applicative definitions for which we proved the existence of a least model is sufficient to translate Datalog clauses into applicative form. This is due to the restrictions of Datalog compared to Prolog, which has terms of unlimited complexity. Our framework, however, translates Prolog just as easily as Datalog.

Our work is also related to higher-order logic programming, as in λProlog [18], HiLog [4], and higher-order Horn logic programming [24], the latter being the closest. We share with these the objective of having relations as first-class citizens, but our approach differs by staying within a first-order logic; see the conclusions.

An interesting related work is found in [17], where McCabe aimed, among other things, at a combined applicative functional and relational framework. He suggested a restricted form of mathematical *set abstractions* to represent relations. This form is a direct extension to logic programming of Darlington's *absolute set abstractions* for functional programming augmented with unification [9]. McCabe uses *predicate* variables which are higher-order constructs. This leads to restrictions on the use of these variables in order to keep his extension into first-order logic programming[1]. In addition, he relies on mapping the extension into standard logic programming and does not provide an operational semantics for set abstractions. Also, he does not discuss modularity in this extension.

Our `where` expressions are inspired by Nait-Abdallah's ions [1]. The

[1]The proof procedure is still affected, however.

main difference is that Nait-Abdallah's work is syntactic, concentrating on rewriting rules.

Finally, applicative programming constitutes important related work [20], as mentioned in chapter 1.

Chapter 5

Conclusions

5.1 Contributions

The main contribution of this thesis is the development of an applicative framework suitable as basis for applicative relational programming. We have shown how this framework can be used to extend to logic programming the program structuring techniques and the parallel to the functionals and the functional expressions of functional programming, without leaving first-order logic programming nor losing its fixpoint semantics.

Tables for relational programming. We have shown that a table can be a natural supplement to the types of values in logic programming, namely as a set of sets of answer substitutions. We have shown, by means of the `where` operator, how to integrate this concept into logic programming in two forms, as a query on a program, and as a condition part in a clause. This concept of tables is appropriate for logic programs viewed as deductive databases, since it coincides with the notion of relations associated with attributes in the database convention.

A framework of tables and relations. We have defined operations on tables and relations that constitute our framework. The operators are: the *application* operator ":" that maps a relation and a tuple of terms to a table, the *intersection* operator "$\cap$" on tables yielding a table, and the *projection* operator "/" which gets relations back from tables, using a generalized form of projection. We have established basic mathematical properties of these operators. In particular, we have shown that our intersection of tables shares properties with set intersection, and that application and projection are inverses under certain restrictions.

An applicative relational interpretation for Horn clauses. We have shown that our framework constitutes an applicative counterpart for Horn clauses, and one operating on whole relations (and, hence, tables). This is reflected in the simplicity of translating Horn clauses into applicative form and vice versa; this proves the existence of the least model for our applicative transformations.

Relations as first-class citizens. One of the powerful paradigms in computing is "functions as first-class citizens" [20]. This in reaction to languages where functions are more restricted in their use than, say, numbers. Certain functional programming languages have demonstrated the advantages of having functions as first-class objects. Prolog, the only relational programming language, does not have relation-valued expressions of any kind. We have shown that our framework provides for relational expressions, relational variables, and even *relationals*, in an analogy to functionals. For example, we have shown an applicative expression that can replace the (non-logical) `setof` primitive of Prolog, while being more concise. An example of

a relational is

$$\lambda q.\lambda r.(X,Z)/(q:(X,Y)\cap r:(Y,Z))$$

which gives the relational composition of any two binary relations. Hence, in our framework, relations are first-class citizens.

Modular logic programming. We have shown how to support modularity and local scoping for relations, which is a deficiency in declarative logic programming with respect to developing large programs. For declarative logic programming, we have presented an apparatus to provide locality for each predicate as goal in a query or as condition in a clause of a program; the declarative Horn clause formalism does not provide locality for predicate names. For applicative logic programming, we have devised a mechanism that facilitates the exportation and importation of relations between programs. This provides for modules that are generic, and even composite, like

$$\lambda ord1.\lambda ord2.Cart'$$

of the composite ordering example.

Higher-order logic is not needed, at least not to provide the desirable programming features implied by having functions and relations as first-class citizens. In functional programming, a function is said to be higher-order when it takes a function as argument or produces one as result. A logic is said to be higher-order when one can quantify over function or predicate symbols. These two senses of "higher-order" do not coincide. This is proved by the existence of formalizations of λ-calculus in first-order logic. A constant of logic can denote any individual, including a function. Therefore, a first-order variable, one ranging over individuals, can range over functions.

In that sense, our expressions contain first-order variables that range over relations. This is an important point, as first-order logic is more tractable, theoretically and practically, than higher-order logic. For example, higher order unification, used in λProlog [18], is admitted to be much more complex than standard unification, and also to be undecidable in general.

5.2 Future work

The major step to follow this work is the development of an applicative relational programming language based on our framework. That would be a twofold achievement since Horn clause form can be incorporated as sublanguage via the `where` operator, thus making both the declarative and applicative styles available within the same language.

The discussions in chapter 3 shed some light on this goal, motivating a definition of such a language, and some of its requirements and features. For example, providing fixpoint semantics for such a language should be possible following the view of recursive applicative definitions as recursive program schemes in the sense of de Bakker and de Roever [10]. A first approach to such a language might legitimately restrict tables to be ground, thus making complete our proof of the existence of the least fixpoint for recursive applicative definitions. This can be achieved by incorporating ξ in the definitions of ":" and `where`.

As for implementation, a first approximation could be a preprocessor for applicative definitions into Horn clause form, across the lines of the translation scheme in appendix A, then using a logic programming system. However, the inference machine of existing Prolog is sufficient indeed to realize applicative definitions. This is because all our operators are based on unification,

while the `where` operator is based on SLD-resolution. The necessary collect-all-answers feature is not readily available in standard Prolog systems which supply one answer at a time. Yet, this facility is available in the TupiLog system [6] mentioned in chapter 3, and can be implemented as well in Or-parallel logic programming languages like Aurora. The latter implementation would be preferred for reasons of efficiency.

The properties of our framework suggest its formalization as an algebra. This is best based on Tarski's homomorphism from first-order predicate logic formulas (open or closed) to cylindric set algebras [12]. Cylinders, then, would be a mathematical model for tables.

An area that can be pursued then is the view of our framework as a relational algebra for databases, especially deductive ones. Traditionally, an algebra is used to express execution alternatives, for query optimization. Algebraic equivalences play an important role in generating alternatives. The usefulness of our framework in this respect is yet to be demonstrated. Nevertheless, the results of chapter 2 constitute a good starting point. For example, the identity

$$\sigma_E(R) \bowtie S = R \bowtie \sigma_E(S)$$

is expressed, according to appendix B, by

$$(R \cap \{E'\}) \cap S = R \cap (S \cap \{E'\}),$$

where E' is a set of the equations in the conjunction of equations E. This clearly holds in our framework by the commutativity and associativity of "$\cap$", both of which are examples of algebraic equivalences. Similarly, the proof of

$$\pi_X(R) \cup \pi_X(S) = \pi_X(R \cup S)$$

in our framework, which is

$$R[X] \cup S[X] = (R \cup S)[X],$$

is a simple task.

Yet, for the purpose of accommodating Datalog, our framework provides a uniform and simple translation that is similar to the normal forms of relational algebras. Our translations of Datalog are related to the Union-Join normal forms for Datalog algebraization in [3], despite the considerable complexity of that algebraization compared to our applicative translations. They are also related to the normal forms of the Project-Select-Join and the Union-Project-Select-Join languages of [22]. These normal forms are important as the end products of algebraic simplification.

The integration of an applicative relational language with the functional formalism is another subject of interesting potential. We have demonstrated the program structuring benefits of this integration. An example of the usefulness of exploiting the techniques of functional programming languages is rule 9 of the translation scheme, in appendix A, which handles relational *abstractions* as *closures*.

A somewhat ambitious goal would be to investigate applicative expressions in general, not only as a vehicle to algebraize Horn clauses. A supporting argument, quoted from [13], is that the

> embedding of the relational algebra in a cylindric set algebra gives a precise and natural meaning for unrestricted relational expressions.

This is shown, in appendix B, to be a point in which our framework, when formalized, would be superior over traditional database relational algebra.

Of course, there are other issues of importance in logic programming that need to be pursued, for example, the method of evaluation appropriate for the suggested system, and the effect(s) of incorporating negation. This complex of points for further study comes as no surprise since the subject is relatively fresh in logic programming and has close relations to other programming paradigms, namely functional and database programming.

Bibliography

[1] M.A. Nait Abdallah. Ions and local definitions in logic programming. In *Theoretical Aspects of Computer Science*, pages 60–72. Springer-Verlag, 1986. Springer Lecture Notes in Computer Science 210.

[2] Rajiv Bagai, Marc Bezem, and M.H. van Emden. On downward closure ordinals of logic programs. *Fundamenta Informatica*, 13(1), 1990.

[3] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[4] W. Chen, M. Kifer, and D.S. Warren. Hilog: A first-order semantics for higer-order logic programming constructs. In *Logic Programming: Proceedings of the North American Conference*, pages 1090–1114, Cleveland, Ohio, 1989.

[5] M.H.M. Cheng. *Lambda-equational Logic Programming.* PhD thesis, University of Waterloo, 1987.

[6] M.H.M. Cheng, M.H. van Emden, and J.H.M. Lee. Tables as a user interface for logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 784–791, Tokyo, Japan, November–Decemeber 1988. Ohmsha, Ltd.

[7] M.H.M. Cheng, M.H. van Emden, and B.E. Richards. On Warren's method for functional programming in logic. Technical Report DCS-122-IR, University of Victoria Logic Programming Laboratory, 1989.

[8] Alain Colmerauer. Prolog and infinite trees. In *Logic Programming*, pages 231–251. Academic Press, 1982.

[9] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Relations, Functions and Equations*, pages 37–70. Prentice Hall, 1986.

[10] J.W. de Bakker and W.P. de Roever. A calculus for recursive program schemes. In M. Nivat, editor, *Automata, Languages and Programming*, pages 167–196. North Holland, 1973.

[11] P.R. Halmos. *Naive Set Theory.* Springer-Verlag, 1974.

[12] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras, Part I*, volume 64 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1971.

[13] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras. *Journal of Computer and Systems Sciences*, 28, 1984.

[14] R.A. Kowalski. *Logic for Problem-Solving.* Elsevier North-Holland, 1979.

[15] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[16] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions of Programming Languages and Systems*, 4:258–282, 1982.

[17] Francis Gregory McCabe. *Logic and Objects.* PhD thesis, Department of Computing, Imperial College of Science and Technology, 1989.

[18] D. Miller and G. Nadathur. Higer-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, 1986.

[19] B.E. Richards. Contributions to functional programming in logic. Master's thesis, University of Victoria, 1990.

[20] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey approach to Programming Language Theory.* MIT Press, 1977.

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

[22] Sieger van Denneheuvel. *Constraint Solving on Database Systems.* PhD thesis, Universiteit van Amsterdam, 1991.

[23] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[24] W.W. Wadge. Higher-order Horn logic programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 289–303. MIT Press, 1991.

Appendix A

Translation scheme

First, we have to evaluate any λ-applications in the given expression. In such cases, where parts of other programs are imported into the given program, we have to standardize, by renaming consistently, the predicate symbols and relational variables in the imported parts apart from those in the given program. This means renaming the predicate symbols in the query operand of `where` according to those in its program operand, since the query is local to that program.

Translation commences by embracing the program, which is considered a (perhaps long) string of characters, with square brackets. Then we repeatedly use a rule, in the translation scheme, whose left-hand side form matches a bracketed substring, in the current string, to transform that substring according to the right-hand side of the rule. Translation ends when there is no bracketed substring left in the program (string).

The following notation is used in the rules of the translation scheme.

- By a relational clause we mean either a relational inclusion or a Horn clause, and by a relational program we mean either an applicative program or a definite program.

- $[X] ::= Y$ means that X transforms into Y.
- X' denotes renaming of X, if X is a predicate symbol, or all its predicate symbols and relational variables, if X is a relational clause, by standardization.
- A, B denotes the conjunction of goals A and B.
- $C; D$ denotes the conjunction of clauses C and D.
- $\langle X \rangle$ denotes $C'_1; \ldots; C'_n$, where $X = \{C_1; \ldots; C_n\}$ is a relational program. That is, $\langle X \rangle$ denotes the substitution of the program X by its clauses, after standardizing its predicate symbols and relational variables apart from the translated program; i.e. the importation of the clauses of X.
- $+X$ denotes appending string X to (the end of) the current string.

Translation scheme:

1. $[\{C_1; \ldots; C_n\}] ::= \{[C_1; \ldots; C_n]\}$, where each C_i is a relational clause.
2. $[C_1; \ldots; C_n] ::= [C_1]; \ldots; [C_n]$, where each C_i is a relational clause.
3. $[C] ::= C$, where C is a definite Horn clause.
4. $[A \leftarrow T] ::= A \leftarrow [T]$, where A is an atom and T is a table.
5. $[(T_1 \cap \cdots \cap T_n)] ::= [T_1], \ldots, [T_n]$, where each T_i is a table.
6. $[\top] ::= []$; i.e. the empty string.

7. $[((\leftarrow A_1, \ldots, A_n)\ \texttt{where}\ R)] ::= A'_1, \ldots, A'_n + [\langle R \rangle]$, where each A_i is an atom and R is a relational program.

8. $[(r : (t_1, \ldots, t_n))] ::= r(t_1, \ldots, t_n)$, where r is a relational variable and $t_1, \ldots, t_n$ are terms.

9. $[(j : (t_1, \ldots, t_n))] ::= [(r : (t_1, \ldots, t_n))] + [r = j]$, where j is a projection, $t_1, \ldots, t_n$ are terms, and r is a new relational variable.

10. $[((\nu r.R) : (t_1, \ldots, t_n))] ::= r'(t_1, \ldots, t_n) + [\langle R \rangle]$, where R is a relational program, r is a relational variable and $t_1, \ldots, t_n$ are terms.

11. $[r \supseteq j_1 \cup \cdots \cup j_n] ::= r[j_1]; \ldots; r[j_n]$, where r is a relational variable and $j_1, \ldots, j_n$ are projections.

12. $[((t_1, \ldots, t_n)/T)] ::= (t_1, \ldots, t_n) \leftarrow [T]$, where $t_1, \ldots, t_n$ are terms, and T is a table.

13. $[r = j] ::= r[j]$, where r is a relational variable and j is a projection.

Example. As an example of the application of the translation rules, we translate the following applicative expression:

$$(\lambda n.Q)((Y)/((\leftarrow \mathrm{num}(Y))\ \texttt{where}\ P_{\mathrm{num}})),$$

where Q is the relational program

$$\{leq \supseteq ((0, X)/(n : (X))); leq \supseteq ((s(X), s(Y))/(leq : (X, Y))\},$$

and P_{num} is the definite program

$$\{\texttt{num(0)} \leftarrow; \texttt{num(s(X))} \leftarrow \texttt{num(X)}\}.$$

Now, carrying out the λ-application yields

$$\{leq \supseteq ((0,X)/(((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}})) : (X)));$$
$$leq \supseteq ((s(X),s(Y))/(leq : (X,Y))\}.$$

Translation of this program, in a left to right manner, proceeds as follows, where an application of a translation rule is indicated by $\stackrel{n}{\longmapsto}$ with n being the rule number.

$\stackrel{1}{\longmapsto} \{[leq \supseteq ((0,X)/(((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}})) : (X)));$
$\quad leq \supseteq ((s(X),s(Y))/(leq : (X,Y))]\}$

$\stackrel{2}{\longmapsto} \{[leq \supseteq ((0,X)/(((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}})) : (X)))];$
$\quad [leq \supseteq ((s(X),s(Y))/(leq : (X,Y))]\},$

of which we translate the first inclusion, as it is more complicated, as follows

$\stackrel{11}{\longmapsto} leq[((0,X)/(((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}})) : (X)))]$

$\stackrel{12}{\longmapsto} leq(0,X) \leftarrow [(((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}})) : (X))]$

$\stackrel{9}{\longmapsto} leq(0,X) \leftarrow [(num : (X))]; [num = ((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}}))]$

$\stackrel{8}{\longmapsto} leq(0,X) \leftarrow num(X); [num = ((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}}))]$

$\stackrel{13}{\longmapsto} leq(0,X) \leftarrow num(X); num[((Y)/((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}}))]$

$\stackrel{12}{\longmapsto} leq(0,X) \leftarrow num(X); num(Y) \leftarrow [((\leftarrow \mathrm{num}(Y)) \ \mathtt{where}\ P_{\mathrm{num}})]$

$\stackrel{7}{\longmapsto} leq(0,X) \leftarrow num(X); num(Y) \leftarrow num'(Y);$
$\quad [num'(0) \leftarrow; num'(s(X)) \leftarrow num'(X)]$

$\stackrel{2}{\longmapsto} leq(0,X) \leftarrow num(X); num(Y) \leftarrow num'(Y);$
$\quad [num'(0) \leftarrow]; [num'(s(X)) \leftarrow num'(X)]$

$\stackrel{3}{\longmapsto} leq(0,X) \leftarrow num(X); num(Y) \leftarrow num'(Y);$
$\quad num'(0) \leftarrow; [num'(s(X)) \leftarrow num'(X)]$

$$\stackrel{3}{\longmapsto} leq(0,X) \leftarrow num(X); num(Y) \leftarrow num'(Y);$$
$$num'(0) \leftarrow; num'(s(X)) \leftarrow num'(X).$$

Notice how rule 9 handles a projection (an anonymous relation) in a way that is similar to that of implementing a λ-abstraction (an anonymous function) in functional programming, namely as a closure. Notice also that even though we choose *num* as name for the new variable in the step using rule 9, the standardization of predicate and variable names in rule 7 took care of the prospective clashes, by considering the locality of `where`'s query operand to its program operand.

Appendix B

Realizing Codd's relational operators

Since our tables are sets (of sets of solved form equations), realizing the set-theoretic operations is trivial; e.g., union and difference. We show how to realize the special operators of Codd's relational algebra in our framework.

For each operator, we write its database form on the left-hand side of an equation with the right-hand side being the equivalent applicative definition. We use A and B for attributes, X and Y for attribute sets, a for an individual object in the domain $D(A)$ of an attribute A, and T and S for tables; Also, (X) means a tuple of elements of X.

1. Natural join: $T \bowtie S = T \cap S$.

2. Equi-join: $T[A = B]S = T \cap \{\{A = B\}\} \cap S$.

3. Projection: $\pi_X(T) = T[X]$.

4. Cartesian product: $T \times S = T \cap S'$, where S' is obtained from S by standardizing τ_S apart from τ_T. The renaming function is simply

defined in our framework as

$$\delta_B^A(T) \stackrel{\text{def}}{=} ((\tau_T)/T) : (\tau_T)\{A = B\}$$

which renames attribute A to B in table T. Hence, S' is $\delta_X^{\tau_S}(S)$, where X is a set of new variables.

5. Selection: $\sigma_E(T) = T \cap \{E'\}$, where the selection condition E is a conjunction of equations of the form $A = B$ or $A = a$, and E' is a set of the equations in E. A more complex condition is possible by allowing disjunction and negation in building the selection condition.

Notice that we did not insist on the usual restrictions on the applicability of these operations; e.g. that $\tau_T = \tau_S$ for $\bowtie$ to be applied. This is because our operators are general enough to elevate these restrictions and our tables can contain variables as entries. This is an aspect in which our framework, once formalized, would be superior to traditional database relational algebras.

VITA

Surname: Ibrahim
Given Names: Husain Ibrahim Haji
Place of Birth: Kuwait, Kuwait
Date of Birth: 21 July 1961

Educational Institutions Attended:

University of Victoria	1990 to 1992
King Abdul Aziz University, Jeddah	1986 to 1990

Degrees Awarded:

B.Sc. (Honours)	King Abdul Aziz University	1990

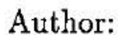
Honours and Awards:

University of Victoria Fellowship	1990-1992
Graduate Teaching Award	1990-1991
Prince Majid Award	1990
Certificate of Excellence	1986-1990

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Applicative Expressions for Relational Programming



Author:

HUSAIN IBRAHIM HAJI IBRAHIM

(Name in Block Letters)

September 16, 1992

(Date)